深入解析HyperLedger Fabric区块链平台使用方案
手把手教你快速部署应用服务，助力开发成功落地

Broadview®
www.broadview.com.cn

# HyperLedger Fabric 开发实战

## 快速掌握区块链技术

杨毅 编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

**杨毅**，9年软件系统开发经验，阿里云MVP，先后从事Android、iOS、Java后台服务及区块链相关开发，获得过多项国家专利，上线过多个基于区块链的项目。目前专注于区块链技术应用及落地场景等方向。

作者博客：
https://www.cnblogs.com/aberic

作者微信订阅号：

# HyperLedger Fabric 开发实战

## 快速掌握区块链技术

杨毅 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

## 内 容 简 介

本书系统地介绍了超级账本 HyperLedger Fabric v1.1 架构的设计和应用方法，包括环境及源码部署、Solo 多机部署、Kafka 集群部署、智能合约编写等。同时，针对第三方可插拔式插件 CouchDB 实战应用，Java-SDK 的应用、编写方案和具体接口执行策略进行了详细讲解。另外，本书以搭建一个反欺诈区块链平台项目为例进行了实战演练，读者可以快速掌握区块链技术。

本书适合区块链系统开发人员阅读，需要有一定的面向对象语言的基础，也可供对开发区块链系统感兴趣的高校师生参考。



**图书在版编目（CIP）数据**

HyperLedger Fabric 开发实战：快速掌握区块链技术 / 杨毅编著. —北京：电子工业出版社，2018.6
ISBN 978-7-121-34173-1

Ⅰ. ①H… Ⅱ. ①杨… Ⅲ. ①电子商务－支付方式－研究 Ⅳ. ①F713.361.3

中国版本图书馆 CIP 数据核字（2018）第 099665 号

责任编辑：宋亚东
印　　刷：天津千鹤文化传播有限公司
装　　订：天津千鹤文化传播有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张 18.5　字数：388 千字
版　　次：2018 年 6 月第 1 版
印　　次：2018 年 6 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

# 前　言

HyperLedger Fabric 最初是由 Digital Asset 和 IBM 公司贡献的、由 Linux 基金会主办的一个超级账本项目，它是一个目前非常流行并广为人知的区块链网络框架的实现方案。作为一种基于模块化架构开发应用程序或解决方案的基础，HyperLedger Fabric 支持如共识和会员服务等即插即用的组件。HyperLedger Fabric 利用容器技术来运行称为“chaincode”的智能合约，该合约包含了系统的应用程序逻辑。

## 为什么写作本书

区块链由于去中心化、开放性、自治性、信息不可篡改及匿名性等特征而受到广泛关注，且目前正处在上升势态。抛开炒作的代币项目，应用于行业联盟链或直接搭建私链的项目，采用 HyperLedger Fabric 作为底层平台无疑是最好的选择之一。

本人从接触 HyperLedger Fabric 项目以来，经历了其 0.6 版本到 1.1 版本的数次迭代。因为早期中文资料稀缺，并且 0.6 版本到 1.0 版本是一个跨度非常大的迭代，导致早期的大部分部署和应用经验失去作用，不得已再次从 1.0 版本开始从头学习。当时中文资料极为有限，且大多数都是单篇翻译或纯粹的概念讲解，导致我一直没有找到入门的头绪，只能不断地从官方文档中汲取知识，并成功搭建了基于 Kafka 类型的集群网络。

有了集群的经验，加深了自己对 HyperLedger Fabric 整个网络事务流程的理解，并以此为基础顺利搭建了基于 Fabric-SDK-Java 的客户端项目。也就在这个时候，开始有了写相关博客的想法，并在博客园上发布了第一篇博客，也是从零开始系列文章的第一篇，开始介绍自己的开发历程和部署经验，希望能通过这样的方式帮助更多的开发伙伴加入 HyperLedger Fabric 大家庭。再后来又建立了“区块链学习分享”的微信订阅号，也通过微信号建立了一个纯技术讨论分享的 HyperLedger Fabric 等区块链相关交流群，并在这样的机缘下结识了电子工业出版社宋亚东老师，并正式开始编写本书。

在写书之前我一直在梳理博客的内容，为了真实地还原生产场景，还自费租赁了 15 台服务器用于测试。在已有博文的基础上，外加后来编写的新文档，我比较顺利地完成了本书的编写工作，且书中的项目都依托于所租赁服务器来完成演练，每一步都基于真实情况和场景中的操作。在此过程中，自己对 HyperLedger Fabric 也有了新的认识和理解。

本书主要以 HyperLedger Fabric 案例为引，层层深入，从单机单节点到 Solo 多机组网再到 Kafka 集群部署，其中穿插文档讲解加深读者的理解。与一些偏概念性质的区块链教程类书籍不同，我希望能通过本书帮助读者实现基于 HyperLedger Fabric 的区块链实践落地。

## 本书主要内容

本书以干货为主，文档为辅，基于 HyperLedger Fabric v1.1 版本进行讲解。总计 10 章，每章主要内容介绍如下：

第 1 章是基本环境部署，包括内网和外网的不同方案，以及内核处理等。

第 2 章是 HyperLedger Fabric 及环境部署，先是用文档讲述了 Fabric 相关的介绍和主要功能点，接着分析了源码部署和镜像处理方面的问题。

第 3 章带着读者一步步跑通官方的 e2e_cli 案例，并在随后对该案例进行了分析。

第 4 章开始，手动部署一次单机多节点网络。

第 5 章跟随前章的脚步，手动部署一次 Solo 多机网络环境。

第 6 章继续深入，搭建基于 Kafka 的集群网络。

第 7 章以文档为主，着重讲解如何编写智能合约及有关智能合约的用法。

第 8 章详细介绍 CouchDB 的使用，并推荐使用 CouchDB。

第 9 章讲解在 Fabric 发布 1.0 版本之后的对外客户端调用方式、客户端对 SDK 的使用和相关源码。

第 10 章以一个简单的案例做演练，在数据链上用到了智能合约，对数据提取则提供了另一种思路。

## 读者对象

这是一本基础讲解类的书，写本书是为了帮助更多的新人入门。所以，本书适合正在寻求 HyperLedger Fabric 入门的新人阅读，也适合中、高级开发人员用作工具书参考。

在阅读本书之前，读者需要具备以下基础知识：

- 具有一定的 Linux 操作系统基本命令的常识。
- 有 Java/Go 等面向对象语言的基础，其中智能合约用 Go 语言编写，SDK 则用到了 Java。如果有这方面的基础，则有助于阅读本书。

## 致谢

在入门及编写本书的时候，有许多人给予了我鼓励和支持。

首先感谢我的妻子，我在开始学习 HyperLedger Fabric 时遇到了很多困难，经常熬夜加班，她始终体谅我，鼓励并支持我。在我写书的时候她不遗余力地帮我查阅相关资料以便我能



够更顺利地完成书稿。

我还要感谢我的领导，也是我的好友王海林，正是他给了我研究学习 HyperLedger Fabric 的机会，并在我遇到困难时帮我逐条分析，厘清思路。他也给予了我在公司最大限度的研发条件和时间，让我在最短的时间里完成了一次自我蜕变。

我还要感谢电子工业出版社的宋亚东老师，感谢他一直以来对我的支持和鼓励，感谢在我早期编写书稿的过程中对内容的编排和规范给予了很多帮助。也感谢电子工业出版社所有参与本书出版工作的老师，是你们的辛勤付出让本书成功出版。

最后，我要感谢我博客的读者及微信群里的朋友们，正是与你们一次次地沟通和探讨，让我不断提升自我，也鞭策我不断前行。

由于我水平有限，书中不足及错误之处在所难免，敬请专家和读者给予批评指正。

杨毅

2018 年 5 月

## 读者服务

轻松注册成为博文视点社区（www.broadview.com.cn）用户，扫码直达本书页面。

- **下载资源**：本书提供示例代码及资源文件，均可在“下载资源”处下载。
- **提交勘误**：您对书中内容的修改意见可在“提交勘误”处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方“读者评论”处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/34173

# 目　录

第 1 章　基本环境部署 ..... 1
1.1　环境整理 ..... 1
1.2　Docker 安装 ..... 2
1.2.1　卸载旧版本 ..... 3
1.2.2　在线安装 Docker CE ..... 3
1.2.3　离线安装 Docker CE ..... 5
1.2.4　Docker 启动及常用命令 ..... 5
1.3　Docker-Compose 安装 ..... 6
1.3.1　在线安装 Docker-Compose ..... 6
1.3.2　离线安装 Docker-Compose ..... 7
1.4　Go 语言环境安装 ..... 8
1.4.1　下载 Go 语言包 ..... 8
1.4.2　配置 Go 语言环境变量 ..... 9
1.5　本章小结 ..... 9
第 2 章　Fabric 及环境部署 ..... 10
2.1　Fabric 介绍 ..... 10
2.1.1　什么是区块链 ..... 10
2.1.2　区块链的作用 ..... 12
2.1.3　超级账本是什么 ..... 14
2.2　Fabric 功能汇总 ..... 16
2.3　Fabric 组成模型 ..... 17
2.3.1　资产 ..... 18
2.3.2　智能合约 ..... 18
2.3.3　账本特征 ..... 18
2.3.4　隐私频道 ..... 19
2.3.5　成员安全性 ..... 20



2.3.6　共识机制 ..... 20
2.4　Fabric 环境部署 ..... 20
2.4.1　Fabric 源码安装 ..... 20
2.4.2　下载 Fabric 镜像 ..... 22
2.4.3　镜像备份和迁移 ..... 26
2.5　本章小结 ..... 28
第 3 章　End-2-End 案例 ..... 29
3.1　平台特定文件 ..... 29
3.2　运行 e2e_cli ..... 31
3.3　e2e_cli 案例分析 ..... 38
3.3.1　容器服务脚本 ..... 38
3.3.2　容器启动配置文件 ..... 52
3.3.3　Fabric 网络解析 ..... 55
3.4　本章小结 ..... 62
第 4 章　部署单机多节点网络 ..... 64
4.1　生成证书文件 ..... 65
4.2　部署 Orderer 节点 ..... 69
4.3　部署 peer0.org1 节点 ..... 70
4.4　搭建 Fabric 网络 ..... 75
4.5　初步接触智能合约 ..... 78
4.6　部署 peer0.org2 节点 ..... 84
4.7　本章小结 ..... 88
第 5 章　Solo 多机部署 ..... 89
5.1　网络拓扑 ..... 89
5.2　部署 Orderer 节点 ..... 91
5.3　部署 peer0.org1 节点 ..... 92
5.4　部署 peer1.org1 节点 ..... 97
5.5　部署 peer0.org2 节点 ..... 101
5.6　本章小结 ..... 107

第 6 章 Kafka 集群部署 ..... 108
6.1 Fabric 账本 ..... 108
6.2 事务处理流程 ..... 110
6.2.1 客户端发起事务 ..... 111
6.2.2 验证签名并执行事务 ..... 112
6.2.3 检查返回协议 ..... 112
6.2.4 客户端将背书合并到交易中 ..... 113
6.2.5 提交并验证事务 ..... 113
6.2.6 账本更新 ..... 114
6.3 读写集规则 ..... 114
6.4 Kafka 集群配置 ..... 116
6.4.1 crypto-config.yaml 配置 ..... 119
6.4.2 configtx 配置 ..... 121
6.4.3 Zookeeper 配置 ..... 125
6.4.4 Kafka 配置 ..... 127
6.4.5 Orderer 配置 ..... 132
6.5 启动集群 ..... 138
6.5.1 启动 Zookeeper 集群 ..... 138
6.5.2 启动 Kafka 集群 ..... 140
6.5.3 启动 Orderer 集群 ..... 144
6.6 集群环境测试 ..... 146
6.7 本章小结 ..... 158
第 7 章 智能合约 ..... 159
7.1 智能合约概述 ..... 159
7.2 背书策略 ..... 160
7.3 使用智能合约 ..... 161
7.3.1 智能合约是什么 ..... 161
7.3.2 智能合约的生命周期 ..... 161
7.3.3 Packaging（包） ..... 162
7.3.4 创建 package（包） ..... 162
7.3.5 包签名（Package signing） ..... 163
7.3.6 安装智能合约 ..... 164
7.3.7 智能合约实例化 ..... 164



7.3.8　升级智能合约 ..... 165
7.3.9　停止及启动智能合约 ..... 166
7.3.10　CLI（客户端） ..... 166
7.3.11　系统智能合约 ..... 168
7.4　编写智能合约 ..... 168
7.4.1　开发人员眼中的智能合约 ..... 168
7.4.2　智能合约接口 ..... 169
7.4.3　一个简单的资产智能合约 ..... 169
7.5　加密智能合约 ..... 178
7.6　系统合约插件 ..... 180
7.7　智能合约 API ..... 182
7.8　Peer 节点与合智能约 ..... 184
7.8.1　安装智能合约 ..... 185
7.8.2　实例化智能合约 ..... 187
7.8.3　调用智能合约 ..... 188
7.8.4　列出智能合约 ..... 190
7.8.5　打包智能合约 ..... 191
7.8.6　查询智能合约 ..... 192
7.8.7　签名智能合约包 ..... 193
7.8.8　升级智能合约 ..... 194
7.9　本章小结 ..... 196
第 8 章　CouchDB ..... 197
8.1　CouchDB 介绍 ..... 197
8.2　启动部署 ..... 201
8.3　索引应用 ..... 206
8.4　查询应用 ..... 216
8.5　选择器语法 ..... 218
8.5.1　基本语法 ..... 218
8.5.2　嵌套对象 ..... 219
8.5.3　运算符 ..... 219
8.5.4　隐式运算符 ..... 220
8.5.5　显示运算符 ..... 222
8.6　本章小结 ..... 226

第 9 章 Java-SDK 客户端 ..... 227
9.1 SDK 项目前置条件 ..... 227
9.2 SDK 代码使用 ..... 232
9.2.1 Orderers 对象 ..... 233
9.2.2 Peers 对象 ..... 235
9.2.3 Chaincode 对象 ..... 238
9.2.4 FabricUser ..... 240
9.2.5 FabricStore ..... 245
9.2.6 FabricOrg ..... 250
9.2.7 FabricConfig ..... 256
9.2.8 ChaincodeManager ..... 257
9.3 SDK 使用方法 ..... 264
9.4 本章小结 ..... 269
第 10 章 项目演练 ..... 270
10.1 反欺诈系统 ..... 271
10.1.1 需求分析 ..... 271
10.1.2 编写合约 ..... 272
10.1.3 线上验证 ..... 278
10.3 本章小结 ..... 283



# 第 1 章　基本环境部署

HyperLedger Fabric 是一个基于模块化架构的分布式账本解决方案平台，它拥有深度加密、便捷扩展、部署灵活及可插拔等特性。它的设计初衷是支持不同组件的可插拔实现，并适应整个经济生态系统中存在的复杂性和高精度性。

与其他区块链平台解决方案相比，HyperLedger Fabric 提供了一种独特的扩展便捷和灵活部署的架构。它更多地适用于联盟链形式，即适合企业级之间的区块链联盟方向，建立在可信任的基础上。如果是企业级区块链部署，建议采用 HyperLedger Fabric 提供的方案。

首次接触 HyperLedger Fabric 开发的用户可以从下文的具体环境部署开始，以了解自己今后所要用到的技术和待加强学习的部分。

在本书中用到的宿主机环境是 Centos，版本为 Centos.x86_64 7.4，内核版本为 4.15，相关配置如下。

```
Distributor ID:    CentOS
Description:  CentOS Linux release 7.4.1708 (Core)
Release:      7.4.1708
Codename:     Core

Linux VM_139_63_centos 4.15.11-1.el7.elrepo.x86_64
```

上述选择并非唯一，如常用的 RedHat 及 Ubuntu 的发行版，包括 MacOS 都可以作为 Fabric 的运行环境支持。

## 1.1　环境整理

Fabric 的节点通过 Docker 容器来运行，启动 Fabric 网络中的节点需要预先安装 Docker、Docker-Compose 和 Go 语言环境，然后在网上拉取相关的 Docker 镜像，再通过配置 Compose 文件来启动各个节点。

如果想让 Docker 在服务器上运行，内核版本不能低于 3.10。如果内核版本不够，则部分功能（如 overlay2 存储层驱动）无法使用，并且部分功能可能不太稳定。

为了成功安装，请首先将 Linux 内核升级到 4.x，并更新本地依赖，以满足 Docker 运行。

当更新本地依赖时，一般执行如下命令即可：

```
sudo yum update
```

如果是在公司服务器内网环境下进行部署，就需要申请 YUM 源 IP 及端口号，具体如下：

```
115.28.122.210:80
112.124.140.210:80
```

该源地址的实际访问域名为 http://mirrors.aliyun.com，此域名 IP 及端口号相对稳定，但也会出现变更的情况。当无法访问阿里 YUM 源所申请对应的 IP/Port 时，请尝试该域名访问，查看其最新 IP/Port 并更新阿里 YUM 的访问权限。

随后可以更新本地 YUM 源，具体操作步骤如下。

步骤 1：备份原来的 YUM 源。

```
sudo cp /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base.repo.bak
```

步骤 2：设置阿里 YUM 的源。

```
sudo wget -O /etc/yum.repos.d/CentOS-Base.repo
http://mirrors.aliyun.com/repo/Centos-7.repo
```

步骤 3：清理缓存并生成新的缓存。

```
sudo yum clean all
sudo yum makecache
```

步骤 4：更新 YUM 库。

```
sudo yum update
```

本操作是为了更新所有的内置库到最新版，因为 Docker 最新版本的安装需要所对应的依赖都是相对较新的版本。为了避免安装依赖的麻烦，故此操作很重要。

---

**注意：**很多人会犯这个错误，以为可以不断地通过手动方式来将各种依赖导入进来，但根本行不通。

在 Linux 环境下如果缺少依赖，可以在 Linux Packages Search 下载并安装，地址为 https://pkgs.org/。

---

## 1.2 Docker安装

Docker 是一个开源的应用容器引擎，也是一个提供混合云上的每个应用程序的容器平台解决方案。如今的企业面临着数字化转型的压力，但它们受到现有应用和基础设施的限制，同



时对日益多样化的云、数据中心和应用程序架构进行了合理化调整。

Docker 在应用程序和基础设施和开发人员之间实现了真正的独立性，并为挖掘其潜力创造了一个更好的协作和创新的模式。让开发者可以将他们的应用及依赖包打包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。容器完全使用沙箱机制，相互之间不会有任何接口。

一个完整的 Docker 由以下几个部分组成：

- Docker Client 客户端。
- Docker Daemon 守护进程。
- Docker Image 镜像。
- Docker Container 容器。

### 1.2.1　卸载旧版本

首先需要对服务器进行清理，如果之前安装过 Docker，需要先执行卸载操作，具体命令如下：

```
sudo yum remove docker \
                docker-client \
                docker-client-latest \
                docker-common \
                docker-latest \
                docker-latest-logrotate \
                docker-logrotate \
                docker-selinux \
                docker-engine-selinux \
                docker-engine
```

**注意：**如果 YUM 提示这些包都没有安装也没有问题。

接下来安装 Docker，而 Docker 的安装需要执行两大步骤，第一步是设置仓库，第二步是安装 Docker CE。

### 1.2.2　在线安装Docker CE

接下来安装所需要的包。yum-utils 提供的 yum-config-manager、device-mapper-persistent-data 和 lvm2 是设备/存储驱动程序所需要的基础应用。具体执行命令如下：

```
sudo yum install -y yum-utils \
device-mapper-persistent-data \
```

```
lvm2
```

使用下面的命令来设置稳定存储库：

```
sudo yum-config-manager \
    --add-repo \
https://download.docker.com/linux/centos/docker-ce.repo
```

可以选择性地启用 edge 和测试存储库，这些存储库包含在 Docker 中，在默认情况下是禁用的。具体执行命令如下：

```
sudo yum-config-manager --enable docker-ce-edge
sudo yum-config-manager --enable docker-ce-test
```

也可以通过使用禁用标志来运行 yum-config-manager 命令，以禁用 edge 或测试存储库。要重新启用它，可以使用-enable 标志。下面的命令禁用 edge 存储库：

```
sudo yum-config-manager --disable docker-ce-edge
```

最后，可执行如下命令安装最新版本的 Docker CE：

```
sudo yum install docker-ce
```

还能通过如下命令安装指定版本的 Docker CE：

```
sudo yum install docker-ce-<VERSION STRING>
```

执行查询 Docker 版本号，看是否安装成功：

```
docker --version
```

正常情况下会出现如下情况：

```
Client:
 Version: 18.03.0-ce-rc4
 API version: 1.37
 Go version:  go1.9.4
 Git commit:  fbedb97
 Built:   Thu Mar 15 07:40:24 2018
 OS/Arch: linux/amd64
 Experimental:false
 Orchestrator:swarm

Server:
 Engine:
  Version:    18.03.0-ce-rc4
  API version:1.37 (minimum version 1.12)
  Go version: go1.9.4
```

```
 Git commit:  fbedb97
 Built:  Thu Mar 15 07:44:03 2018
 OS/Arch:    linux/amd64
 Experimental:    false
```

### 1.2.3　离线安装Docker CE

如果是在公司内网环境下进行部署，则无法通过在线方式进行 Docker CE 的安装，需要从官方下载对应的离线包。

---

**注意**：本书在编写时 Docker 官方的最新版为 18.03.0.ce-1.el7.centos.x86_64，读者可以在官方下载页面下载最新版，地址如下：

https://download.docker.com/linux/centos/7/x86_64/stable/Packages/

---

选择下载指定的版本，可将最新版下载至/tmp/docker/docker 目录下。随后执行如下命令进行安装：

```
cd /tmp/docker/docker
yum install docker-ce-18.03.0.ce-1.el7.centos.x86_64.rpm
y
```

安装完成后，按第 1.2.1 节所述执行查询 Docker 版本号，可以检查是否安装成功。

更多安装方面的细节可以参阅 Docker 官方文档进行操作，地址为 https://www.docker.com/。

### 1.2.4　Docker启动及常用命令

Docker CE 安装完成后，需要启动它并设置为开机启动。

Docker 启动命令如下：

```
service docker start
```

Docker 开机自启动命令如下：

```
chkconfig docker on
```

Docker 常用命令如下。

杀死所有正在运行的容器：

```
docker kill $(docker ps -a -q)
```

删除所有已经停止的容器：

```
docker rm $(docker ps -a -q)
```

删除所有镜像：

```
docker rmi $(docker images -q)
```

强制删除所有镜像：

```
docker rmi -f $(docker images -q)
```

## 1.3 Docker-Compose安装

Compose 是定义和运行多容器 Docker 应用程序的工具，可以使用 YAML 文件来配置应用服务。然后，通过单个命令可以从配置中创建并启动所有服务。

```
要了解更多关于组合的特性，请参阅特性列表：
https://docs.docker.com/compose/overview/#features
```

使用 Compose 基本上是一个三步骤的过程：

（1）用 Dockerfile 定义应用程序的环境，这样它可以在任何地方复制。

（2）通过 docker-compose.yml 在服务中定义所启动的各个应用，这些应用将在相互隔离的环境中同时运行。

（3）运行 docker-compose up，启动 Compose 并运行整个应用程序。

### 1.3.1 在线安装Docker-Compose

安装 Docker-Compose 需要服务器支持 curl 命令，如果服务器不支持 curl，需要执行如下操作安装 curl 依赖：

```
yum install curl
```

通过 https://github.com/docker/compose/releases 网址，可以查看 Docker Compose 最新发行版的动向。

---

**注意**：本书所使用的 Docker Compose 的版本为 1.20.1。

---

执行如下操作下载 Docker Compose：

```
sudo curl -L https://github.com/docker/compose/releases/download/1.20.1/docker-
compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
```

该下载目录为/usr/local/bin/docker-compose，且权限已经给出，再执行 docker-compose --

version 检查版本号，或许会有如下提示：

```
-bash: /usr/bin/docker-compose: No such file or directory
```

如果出现上述提示，则执行以下操作：

```
cp /usr/local/bin/docker-compose /usr/bin
```

将 docker-compose 复制到/usr/bin 目录下，再次执行：

```
docker-compose --version
```

正常情况下会打印 docker-compose 的版本信息，如下所示：

```
docker-compose version 1.20.1, build 9e633ef
docker-py version: 2.7.0
CPython version: 2.7.13
OpenSSL version: OpenSSL 1.0.1t  3 May 2016
```

### 1.3.2　离线安装Docker-Compose

Docker-Compose 的离线安装过程相对于 curl 安装稍复杂一点，在第 1.3.1 节中提供了 Compose 官方的 GitHub 项目地址，在该项目中下载最新版的 docker-compose-Linux-x86_64，随后可以将其上传至/tmp/docker/docker-compose 下。在编写本书时，官方提供的 docker-compose 最新 RELEASE 版为 1.20.1。

接着执行如下命令完成安装。

进入当前 docker-compose-Linux-x86_64 上传目录：

```
cd /tmp/docker/docker-compose
```

将其移动至/usr/local/bin/安装目录：

```
mv docker-compose-Linux-x86_64 /usr/local/bin/docker-compose
```

赋予可执行权限：

```
chmod +x /usr/local/bin/docker-compose
```

赋予可执行权限是必不可少的，随后执行如下命令查看 docker-compose 版本信息并确认安装是否生效。

```
docker-compose --version
```

## 1.4 Go语言环境安装

Go 是一种开源编程语言，它使得构建简单、可靠和高效的软件变得容易。

Go 语言是 2007 年末由 Robert Griesemer、Rob Pike、Ken Thompson 主持开发的，并于 2009 年 11 月开源，在 2012 年早些时候发布了 Go 1 稳定版本。现在 Go 语言的开发已经是完全开放的，并且拥有一个活跃的社区。

HyperLedger Fabric 的许多组件和结构都是使用 Go 语言编写的。

### 1.4.1 下载Go语言包

Go 语言环境的安装需要一点小技巧，由于其官方网站 https://golang.org/可能无法顺利访问，可以通过其他途径找到其 Linux 版本的下载路径。

---

**注意：** 可以通过 https://storage.googleapis.com/golang/寻找 Go 版本信息并下载。

本书使用了 Go 最新 RELEASE 版为 1.10.1。

---

之前安装 Docker Compose 时已经安装了 curl 命令，可以执行以下操作下载最新版本的 Go 语言包：

```
curl -O https://storage.googleapis.com/golang/go1.10.1.linux-amd64.tar.gz
```

或直接通过 https://golang.org/doc/install?download=go1.10.1.linux-amd64.tar.gz 链接下载最新版。

---

**注意：** 该 URL 中的版本号甚至链接本身以官网为主。

---

如果直接通过地址进行下载，可在下载完成后将其上传至/tmp/docker 目录下。

进入/tmp/docker 目录：

```
cd /tmp/docker
```

解压 go1.10.1.linux-amd64.tar.gz 至/usr/local 目录下，执行如下操作：

```
tar -C /usr/local -xzf go1.8.3.linux-amd64.tar.gz
```

如果是通过 curl 下载的，则可直接执行上述命令进行解压缩操作。

本书上传或指定的安装目录都是非固定的，但注意，如果上传至 tmp 目录，Linux 系统会不定时地清除里面的内容。

### 1.4.2　配置Go语言环境变量

考虑到以后将会在 Go 语言中编写链码程序，有两个环境变量需要正确设置；可以将这些设置永久地保存在适当的启动文件中。

修改/etc/profile 文件使其永久性生效，并对所有系统用户生效，在文件末尾加上如下两行代码：

```
export PATH=$PATH:/usr/local/go/bin
export GOPATH=/opt/gopath
```

上述修改/etc/profile 文件的具体实现操作如下：

```
cd /etc
vim profile
```

执行修改后，继续执行：

```
source profile
```

使其修改生效。随后可通过下述命令查看是否添加成功：

```
echo $PATH
```

最后，可通过如下命令查看当前 Go 版本的信息：

```
go version
```

正常情况下如下所示：

```
[root@VM_139_63_centos ~]# go version
go version go1.10.1 linux/amd64
```

## 1.5　本章小结

通过上述 4 个大步骤的操作，可以准备好 Linux 系统中必要的环境，安装 Docker、Docker Compose 及 Go 语言环境，HyperLedger Fabric 运行的基础条件已经准备完毕。这里再简单回顾上述步骤中的几个要素。

HyperLedger Fabric 是运行在 Docker 容器网络环境下的应用，而 Docker 本身的要求则需要 Linux 系统内核版本在 3.10 以上，在编写本书的时候最新版本 4.15 也可以完美运行。Docker 容器的配置及启动由 Docker Compose 辅助完成，也包括后续章节中将会提到的 HyperLedger Fabric 执行必备镜像等操作。HyperLedger Fabric 源码中的绝大部分组件和结构都是由 Go 语言编写的，所以 Go 语言环境必不可少，同时对 Go 语言的深入学习也可以加强对 HyperLedger Fabric 的了解。

# 第 2 章　Fabric 及环境部署

在正式开始学习 Fabric 之前，需要对 HyperLedger Fabric 有一些简单的认识，本书主要从基本介绍、功能汇总和组成模型三个方面简单地说明什么是 HyperLedger Fabric，也能在查看源码目录时有一个较为清晰的印象。

如果读者对概念性的东西没有兴趣或已经知晓，可以略过前三节，直接从第 2.4 节开始阅读。

## 2.1　Fabric介绍

HyperLedger Fabric 是一个基于模块化架构的分布式账本解决方案平台。本章主要是接续之前的介绍，帮助读者了解区块链的工作原理和 HyperLedger Fabric 特定的特性及组件。

熟悉区块链及 HyperLedger Fabric 的结构原理后，就可以开始正式使用 HyperLedger Fabric 搭建属于自己的平台。

### 2.1.1　什么是区块链

区块链网络的核心是一个分布式账本，记录所有在网络上发生的交易。

在区块链中，账本会被所有网络中的参与者复制到本地，且每一个参与者都在对账本进行维护协作，因此它是完全去中心化的，如图 2-1 所示。

除了去中心化，还使用了加密技术，每一个区块都有唯一的 Hash，即便是通过网络将账本复制到本地应用服务器中，也无法篡改其中的内容。这种不可篡改的特性使得信息具备可追溯的能力，因为所有的参与者在提交信息后都无法改变，都会在区块中留存记录，这也是区块链有时被称作证明系统的原因。

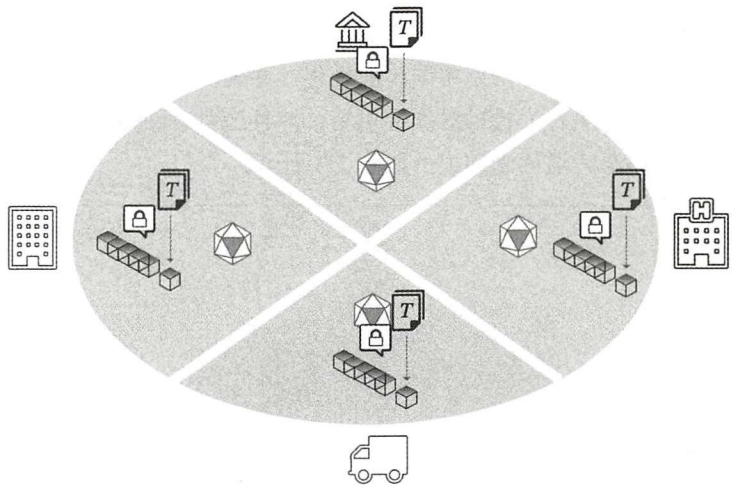

图2–1　去中心化网络

为了支持信息更新的一致性，并支持完整的账本功能（包括但不限于交易、查询等），区块链网络通过使用智能合约来约束和规范对账本的访问及变更，如图 2-2 所示。

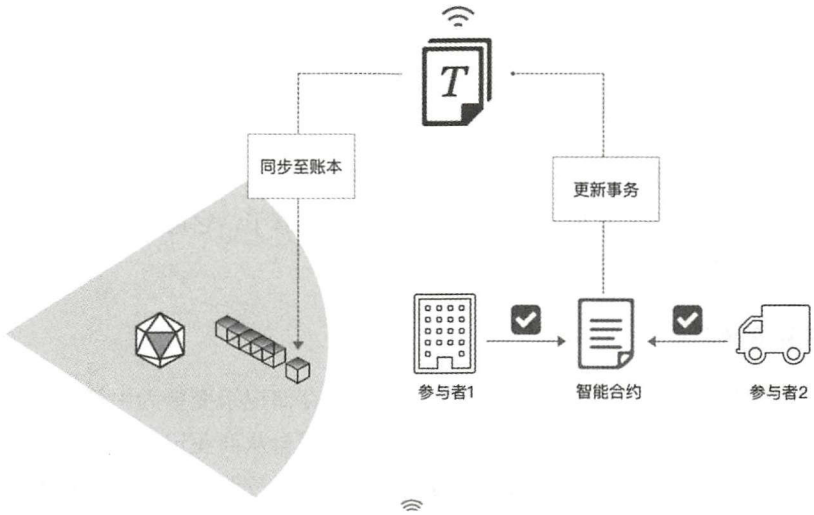


图2–2　智能合约

在智能合约中封装了信息处理的完整方案，以此来尽量简化整个网络的信息交易。智能合约通过编写可以被升级，通过升级来丰富其功能并增强其对事务的处理能力。所有的参与者都可以按照智能合约中的约定自动执行相关事务处理操作。

例如，一份智能合约可以规定货物运输的成本，成本根据货物到达的时间而变化。在双方

同意的前提下，当收到货物时，资金会根据智能合约中的约定自动地转手。

保持账本中发生的交易在整个网络中同步的过程，并确保只有当交易得到拥有决策权力的参与者（背书方或符合背书条件）批准时才会更新，并且当所有网络账本进行更新时，它们以相同的顺序更新相同的事务，这称为共识，如图 2-3 所示。

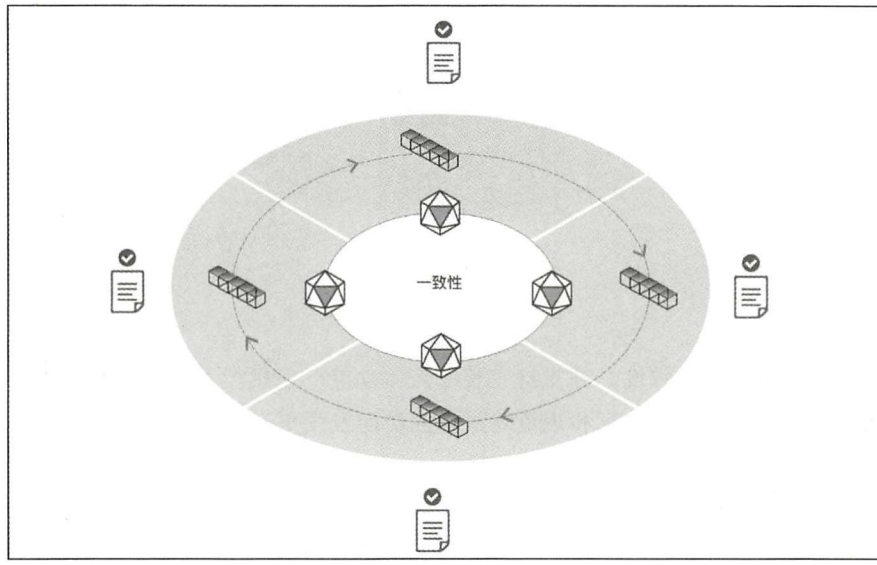


图2-3　共识

通过对区块链的了解更加深入，以后会学到更多关于账本、智能合约和共识的知识。就目前而言，将区块链视为一个共享的、复制的交易系统就足够了，它通过智能合约进行更新，并通过一个叫作共识的协作过程保持一致。

## 2.1.2　区块链的作用

当今的交易网络只是稍微更新的网络版本，即本地存储记录变更为由第三方中介的多账本记录方案（但各家账本内容仅与自身相关），该记录方式自从商业记录被保存以来就已经存在了。业务网络的成员彼此之间进行交易，但他们保持各自的交易记录。而他们所做的事情——无论是 16 世纪的佛兰德挂毯还是今天的证券——都必须在每次出售的时候确定他们的出处，以确保出售某件物品的企业拥有一串头衔（合法记录）来证明他们的所有权。如图 2-4 所示为当前系统记录方案。

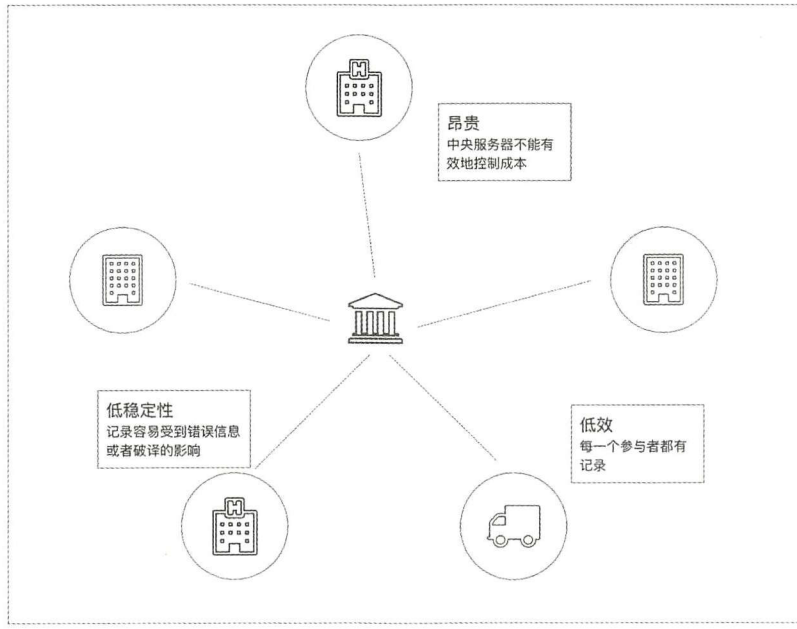


图2-4　当前系统方案

现代科技已经将这一过程从石片、纸质文件升级到硬盘和云平台，但底层结构是一样的。并不存在统一的系统来管理网络参与者的身份，因此对参与者的交易信息建立可靠的来源是一件非常费力且痛苦的事情，就好比证券交易的清理就需要耗费数日的时间来操作（世界上的证券交易量高达数万亿美元）。合同必须手动签署和执行，系统中的每个数据库都包含唯一的信息，因此代表了会出现的一个单点故障。

在当今信息共享的过程中，构建一个跨越商业网络的记录系统是不可能的，尽管可见性和信任的需求是清晰的。

如果由“现代”交易系统替代那些无效率的方式，那么商业网络就有了在网络上建立身份、执行事务和存储数据的标准方法吗？如果想要建立一个资产的来源，并且可以通过查看曾经写过的事务列表，还不允许更改，这个来源就是可以被信任的，又该怎么实现呢？

设想中的商业网络如图 2-5 所示。

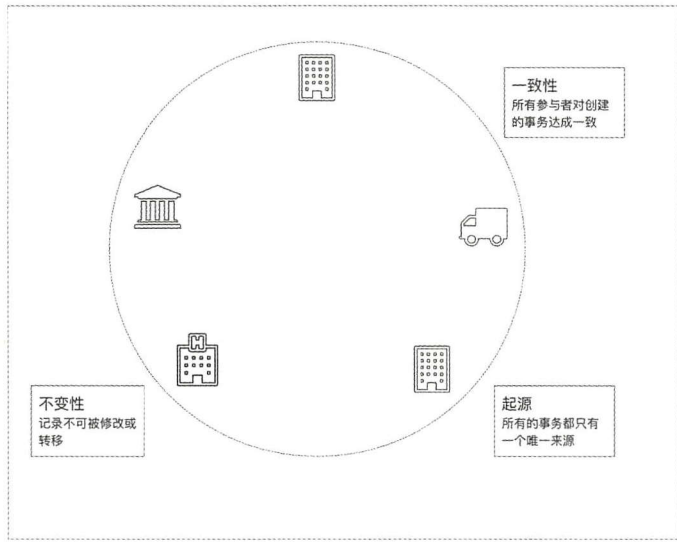


图2-5　设想中的商业网络

这就是一个区块链网络。每一个参与者都有自己本地复制的账本。除了账本信息被共享之外，更新账本的过程也被共享。不同于今天的系统，一个参与者的私人程序被用来更新他们的私人账本，一个区块链系统已经共享程序来更新共享的账本。

由于能够通过共享的账本来协调业务网络，区块链网络可以减少与私人信息和处理相关的时间、成本和风险，同时还能提高信任度和可见度。

通过上文介绍，现在应该大致了解区块链是什么，以及它的作用。还有很多其他重要的细节，但是它们都与信息和过程共享的基本思想有关。

### 2.1.3　超级账本是什么

Linux 基金会在 2015 年创立了 HyperLedger Fabric，以推进跨行业的区块链技术。它没有宣布单一的区块链标准，而是鼓励以一种合作的方式，通过社区进程开发区块链技术，知识产权鼓励开放开发，并随着时间的推移采用关键标准。

HyperLedger Fabric 是 HyperLedger 上的区块链项目之一，如同其他区块链技术一样，它有一个账本，使用智能合约，并且是一个由参与者管理它们的交易的系统。

与其他区块链系统最大的不同点在于 HyperLedger Fabric 是私有的，而且是被许可的。它不是一个开放的无许可的系统，所有参与该网络的成员必须是已确认身份的组织（要求协议验证事务并确保网络的安全），HyperLedger Fabric 组织的成员可以通过一个成员服务提供者（Membership Service Provider，MSP）来注册。

HyperLedger Fabric 还提供了几个可插拔的组件。账本数据可以以多种格式存储，一致的机制可以被转换和输出，并且支持不同的 MSPs。

HyperLedger Fabric 也提供了创建频道（channel）的能力，允许一组参与者创建一个单独的共同维护的交易账本。对于有些参与者可能是竞争对手的网络来说，这是一个特别重要的选择，他们不希望自己的每笔交易都能获得。例如，他们向一些参与者提供了一个特别的价格，而不是其他参与者。如果两个参与者形成一个频道，那么这些参与者及其他参与者都有该渠道的分类账本。

它包括如下四个内容。

1. 共享账本

HyperLedger Fabric 有一个分类子系统，包括两个组成部分：世界状态（world state）和事务日志（transaction log）。每个参与者都有一份账本的副本到他们所属的每一个 HyperLedger Fabric 的网络上。

在给定的时间点上，世界状态组件描述了总账的状态，它是账本的数据库。事务日志组件记录所有导致当前世界状态值的事务。这是世界状态的更新历史。账本是世界状态数据库和事务日志历史的组合。

该账本为世界状态提供了可替换的数据存储方案。在默认情况下，这是一个 LevelDB 键值存储数据库。事务日志不需要是可插拔的，它只是记录了区块链网络使用的账本数据库之前和之后的值。

2. 智能合约

HyperLedger Fabric 的智能合约是用 Chaincode（链码）编写的，并且当应用程序需要与账本进行交互时，被应用程序外部的应用程序调用。在大多数情况下， Chaincode 只与总账的数据库组件交互，例如世界状态（例如查询它），而不是事务日志。

Chaincode 可以用几种编程语言实现，目前支持的 Chaincode 编写的是 Go 语言，在今后的发行版中将会逐步新增 Java 和其他语言的支持。

3. 隐私

根据网络的需要，企业对企业（B2B）网络的参与者可能对他们所共享的信息非常敏感。对其他网络来说，隐私不会成为首要关注的问题。

HyperLedger Fabric 支持需要将隐私（使用频道）作为关键操作需求的网络，同时也是相对开放的网络。

4. 共识

事务必须按照它们发生的顺序写在账本上，即使它们可能是网络中不同的参与者生成的。要做到这一点，必须建立事务的顺序，并且必须在账本中建立一种拒绝错误事务（或恶意的）的方法。

这是一个经过彻底研究的计算机科学领域，有很多方法可以实现它，每一个都有不同的权衡。例如，PBFT（拜占庭式容错）可以为文件副本提供一种机制，使其能够相互通信，从而保持每个副本的一致性，即使是在出现腐化的情况下。或者，在比特币中，排序是通过一个名为“挖矿”的过程来实现的，在这个过程中，竞争的计算机竞相解决一个加密难题，该难题定义了所有流程随后构建的顺序。

HyperLedger Fabric 的设计使得网络启动者可以选择一种最能代表参与者之间关系的共识机制。就像隐私一样，需要有一系列的需求；从人际关系高度结构化的网络到更加对等的网络。

关于 HyperLedger Fabric 共识机制，它目前包括 Solo 和 Kafka，并将很快扩展到 SBFT（简化的拜占庭式容错）。

## 2.2 Fabric功能汇总

Hyperledger Fabric 是一种模块化的区块链架构，是分布式记账技术（DLT）的一种独特的实现，它提供了可供企业运用的网络，具备安全、可伸缩、加密和可执行等特性。Hyperledger Fabric 提供了以下区块链网络功能：

1. 身份管理

为了支持被许可的网络，Hyperledger Fabric 提供了一个成员身份服务（Membership Identity Service），它管理用户 ID 并对网络上的所有参与者进行身份验证。访问控制列表可以通过特定网络操作的授权来提供额外的权限。例如，一个特定的用户 ID 可以被允许调用一个链代码应用程序，但是阻止了部署新的链代码。关于 Hyperledger Fabric 网络的一个真理是，成员相互了解（身份），但他们不知道彼此在做什么（隐私和机密性）。

2. 隐私和机密性

Hyperledger Fabric 使得竞争的商业利益和任何需要私人的、机密的交易的团体能够在同一个被许可的网络上共存。私有频道是受限制的消息传递路径，可用于为网络成员的特定子集提供事务隐私和机密性。所有的数据，包括事务、成员和频道信息，都是不可见的，任何网络成员都不能访问该频道。

3. 高效处理能力

Hyperledger Fabric 通过节点类型分配网络角色。执行事务的操作从事务排序和提交验证中分离出来，以便向网络提供并发性控制和并行性操作。在排序之前，执行事务使每个对等节点能够同时处理多个事务。这种并发执行提高了每个对等点的处理效率，并加速了对排序服务的事务的交付。

除了启用并行处理之外，还可以从事务执行和分类维护的需求中提取节点，而对等节点则从排序（一致的）工作负载中解放出来。角色的这种分支也限制了授权和身份验证所需的处理；所有的对等节点不需要信任所有的排序节点，反之亦然。因此，在一个节点上的进程可以独立于另一个节点进行验证。

4. Chaincode功能

Chaincode 应用程序对频道中特定类型的事务调用的逻辑进行编码。例如，为资产所有权变更定义参数的 Chaincode，确保所有转移所有权的交易都服从相同的规则和要求。系统 Chaincode 是一个特殊的 Chaincode，它定义了整个频道的操作参数。生命周期和配置系统 Chaincode 定义了频道的规则，认可和验证系统 Chaincode 定义了支持和验证事务的需求。

5. 模块化设计

Hyperledger Fabric 实现了一个模块化的架构，为网络设计师提供功能选择。例如，特定的识别、排序（一致）和加密的算法可以被插入到任何一个 Hyperledger Fabric 的网络中。其结果是一个通用的区块链架构，任何行业或公共领域都可以采用，并保证其网络将在市场、监管和地理界线之间进行互操作。

## 2.3 Fabric组成模型

本文主要讲述 Hyperledger Fabric 的关键设计特性，并描述如何实现了一个全面的、可定制的企业级区块链解决方案：

**（1）资产定义。**资产在这里理解为任何具有货币价值的东西，它们都可以通过网络进行交易，无论是超市商品到古董车再到货币期货，都属于资产。

**（2）智能合约。**Chaincode 即 Fabric 的智能合约，分为系统链码和用户链码。链码的执行由事务排序划分，限制了节点类型间的信任和验证级别，并优化了网络的可伸缩性和性能。

**（3）账本特征。**账本不可变的、共享的并且为每个频道编码了整个事务历史，还包含了类似 SQL 的查询功能，用于高效的审计和解决争议。

**（4）隐私频道。**多频道交易的设计方案可以确保竞争的企业和受监管的行业在一个公共网络上交换资产时的高度隐私及保密性。

**（5）成员安全。**Hyperledger Fabric 只允许被授权加盟的成员参与数据维护，且成员间相互认可所有的交易都会被彼此发现和跟踪。这种方式提供了一个可信的区块链网络。

**（6）共识机制。**共识策略的设定是为了达成一致的一种独特的方法，它可以实现企业间所需的灵活性和可伸缩性。

### 2.3.1 资产

资产可以从有形资产（如房地产和硬件）到无形资产（如合同和知识产权）。Hyperledger Fabric 提供了使用智能合约交易修改资产的能力。

资产在 Hyperledger Fabric 中以键-值对集合的形态存在，在频道中各本地账本可以对其状态提交变更事务。资产可以用二进制和（或）JSON 形式表示。

可以通过 Hyperledger Fabric 的 Composer 工具很容易地定义和使用 Hyperledger Fabric 应用程序中的资产。

Composer 工具可以访问 https://github.com/hyperledger/composer 来进行更深入的了解和学习。

### 2.3.2 智能合约

智能合约是定义资产并且可以用于修改资产的事务指令的软件。换句话说，它就是一个频道所有的业务逻辑。智能合约制定了执行读取或修改键值对以及其他状态数据库信息操作的规则。智能合约通过一个事务请求来执行对账本的当前状态数据库操作。智能合约执行会生成一组读写集，这组读写集可以通过网络提交给排序服务节点，并由排序服务节点广播且应用到所有的对等节点上。

### 2.3.3 账本特征

在 Fabric 中产生的所有针对数据状态变更的请求，都会生成有序且不可篡改的记录存于账本中。数据状态的变更是由所有参与方认可的智能合约调用事务的结果。每个事务都将产生一组资产键-值对，这些键-值对作为创建、更新或删除等操作而同步到所有账本。

账本由区块链（区块根据 Hash 等算法组成的链条）组成，而每一个区块中都存储有一条或一组有序的且不可篡改的记录，也就是一个状态数据库来维护当前的 Fabric 的状态。每个频道都有且仅有一个账本，在该频道中的每个加盟成员的对等点都维护同一份账本。

（1）通过使用基键（键查询）、范围查询及组合键查询等方法可对账本执行查询和更新操作。

（2）使用富查询语言的只读查询（如果使用 CouchDB 作为状态数据库）。

（3）只读历史查询通过一个键来查询账本历史记录，支持数据来源场景。

（4）每一条请求的结果都由通过智能合约读取的读集和智能合约写入的写集的键值的多版本组成。

（5）每一条被提交的请求都包含提交该请求的节点的签名证书，并同时提交到排序服务节点。

（6）同一个频道中的区块里的所有请求事务都会被排序，并且这些区块会被排序服务节点广播到该频道内的所有对等节点。

（7）对等节点对请求事务的验证依靠背书策略并严格执行该策略。

（8）在添加一个块之前，执行了版本控制检查，以确保被读取的资产的状态在链代码执行时间之后没有改变。

（9）即将执行变更的请求事务集在新增到一个区块之前必须要做一次版本验证，以确保被读取的资产状态集在本条智能合约执行时间之前没有改变过。

（10）一旦请求事务被验证且提交，就不可篡改。

（11）一个频道的账本包含一个区块生成的配置策略、访问控制列表和其他相关信息。

（12）考虑到频道将会从不同的证书机构得到加密文件，因此频道中拥有成员服务提供者（MSP）实例。

### 2.3.4　隐私频道

Hyperledger Fabric 在每个频道中都有一个不可篡改的账本，以及一个可以操纵和修改当前资产状态的智能合约（例如，更新键-值对）。一个账本限制在一个频道的范围内，它可以在整个网络中共享（假设每个参与者都在一个公共频道上运行），或者它也可以被私有化，只包含一组特定的参与者。

在上述中的后一种情况下，这些参与者将创建一个单独的频道，从而使他们的事务和账本隔离出来。为了满足即公开透明又能保护隐私的场景，智能合约只能在需要访问资产状态来执行读和写操作的对等节点上安装。换句话说，如果一个智能合约没有安装在对等节点上，它将无法调用账本暴露出去的接口。

为了进一步混淆数据，智能合约中的值可以使用诸如 AES 之类的通用加密算法进行加密（在一定程度上或全部使用），然后将事务发送到排序服务，并将生成的区块追加到账本上。一旦加密的数据被写入到账本，它只能被拥有相应的密钥用户解密。

### 2.3.5 成员安全性

Hyperledger Fabric 是一个支持所有参与者都有自己身份的交易网络。公钥的底层方案用于生成与组织、网络组件和最终用户或客户端应用程序绑定的加密证书。因此，可以在更广泛的网络和频道层次上对数据访问控制进行操作和治理。隐私和保密是最重要和最关键的问题，Hyperledger Fabric 这种“被许可”的概念，再加上频道的存在和功能，有助于解决该问题。

### 2.3.6 共识机制

在分布式账本技术中，共识作为单一功能最近成为了一种特定算法的同义词。然而，共识不仅仅是简单地就事务的顺序达成一致，而且在 Hyperledger Fabric 中这种通过它在整个事务流中的基本作用，包括从提交请求、背书验证，到事务排序、确认和广播，这种区别显得尤为突出。简单地说，共识被定义为一个完整的循环，它是由一个经过验证核实的区块所包含的一组事务。

当一个区块中的事务集合的顺序和结果经过所有的检查而符合策略标准时，将最终达成一致。这些检查和平衡发生在一个个请求事务的生命周期中，包括使用背书策略来规定哪些特定的成员必须支持某个事务类，以及系统智能合约，以确保这些策略得到执行和维护。在提交排序服务节点之前，这些执行验证的对等节点将使用这些系统智能合约来确保足够的背书支持，并从适当的实体中获得。此外，在任何包含事务的区块被添加进账本之前，将进行版本控制，在此期间，将对账本的当前状态进行商定或同意。最后的检查提供了对双重开销操作（处理相同事务）和其他可能危害数据完整性的威胁的保护，并允许被执行的方法依赖非静态变量。

除了大量的背书认可、有效性和版本控制检查之外，在事务流的各个方向上也有正在进行的身份验证。访问控制列表是在通过网络实现的（排序服务下发到所有频道），并且因为一条事务请求将会通过不同的体系结构组件，所以请求事务会被反复地签名、复核及验证。总之，达成共识并不仅仅局限于一组交易的达成一致，而是一种包罗万象的特性，它是在交易从提案到最终广播过程中进行的持续验证的副产品。

## 2.4 Fabric环境部署

### 2.4.1 Fabric源码安装

在本书编写时，HyperLedger Fabric 发布的最新稳定版为 1.1RELEASE，所以本书中的所有环境都是围绕该版本进行。

HyperLedger Fabric 的源码由官方在 GitHub 上发布托管，源码的托管地址是 https://github.com/hyperledger/fabric。

需要下载其源码，也将会使用到源码中所提供的案例和工具。工具编译需要用到 Go 语言环境，因此需要将源码目录放置到$GOPATH 路径下。

可以使用 Git 命令下载源码，也可以使用 go get 命令。这里直接用 go get 命令获取最新的 Fabric 源码：

```
go get github.com/hyperledger/fabric
```

由于网络的原因，这一步操作可能需要等得时间比较长，等源码下载完成之后，可以在 ~/go/src/github.com/hyperledger/fabric 中找到所有的最新的源代码。由于 Fabric 一直在更新，所以并不需要最新的源码，需要切换到 v1.1.0 版本的源码即可。具体通过如下命令实现：

```
cd /opt/gopath/src/github.com/hyperledger/fabric/
git checkout -b v1.1.0
```

本步骤也可直接在 GitHub 上将源码下载至本地，再通过 FTP 上传至 hyperledger 目录。

如果没有 Git 命令，还需要先执行以下命令，构建本地 Git 环境。

```
yum install git
```

除了上述方法，也可以更为直接地在 GitHub 上将 Fabric1.1.0 版本的项目直接下载至本地，随后上传至/home 或/data 或其他数据存储的目录中。

这里需要说明的是，Fabric 源码的安装目录并非一定要在指定位置，但在 YAML 中的工作路径和映射路径一定不能写错，否则会导致 Fabric 节点中 shim 的 API 各种调用失败（原因如下所示），所以还是建议按照官方的习惯部署。

```
import (

 "github.com/hyperledger/fabric/core/chaincode/shim"
 pb "github.com/hyperledger/fabric/protos/peer"
)
```

如上所述，本书选择/home 作为根目录，所以 Fabric 源码会安装在/home/docker/github.com/hyperledger/fabric 路径下，不必在意这个路径中的 Docker 目录名称，需注意“github.com/hyperledger/fabric”才是关键。最终的目录结构如图 2-6 所示。

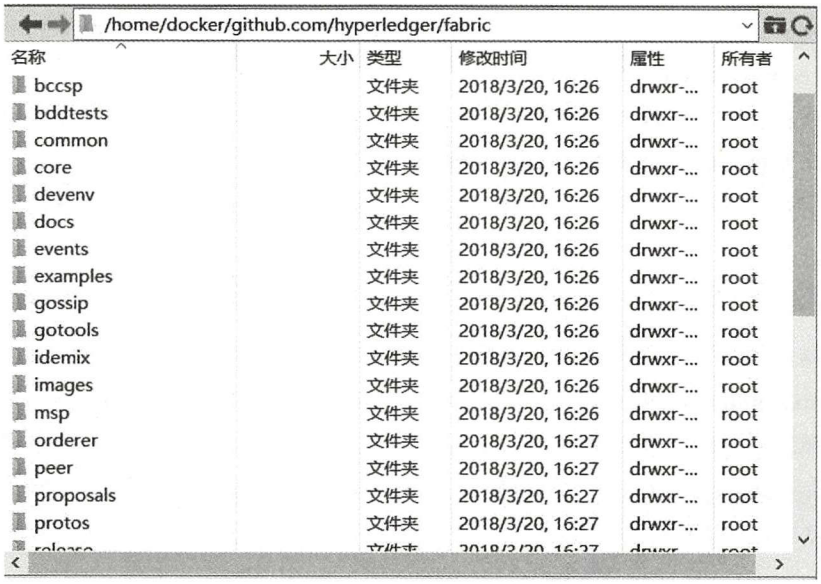


图2-6 目录结构

## 2.4.2 下载Fabric镜像

获取运行 Fabric 所需的镜像获取有多种方式，如果仅仅是测试 Fabric 集群方案，直接进入 fabric/examples/e2e_cli 目录下，运行./download-dockerimages.sh 脚本文件即可下载该工程必要的镜像文件。

一般情况下，为了保证镜像与下载到 HyperLedger Fabric 中的源码 Demo 版本号相对应，该种方法属于较为妥当的方案。

但为了今后升级方便，且版本可以由自己控制，作者还会介绍另一种方案，这也是本书推荐的方案。

所有的 Fabric 相关镜像文件均可以在 DockerHub 官方镜像网站进行遍历和下载，镜像获取地址为 https://hub.docker.com/。检索 HyperLedger 结果如图 2-7 所示，然后再以 hyperledger/fabric-peer 为例，进入其下载页面，如图 2-8 所示。

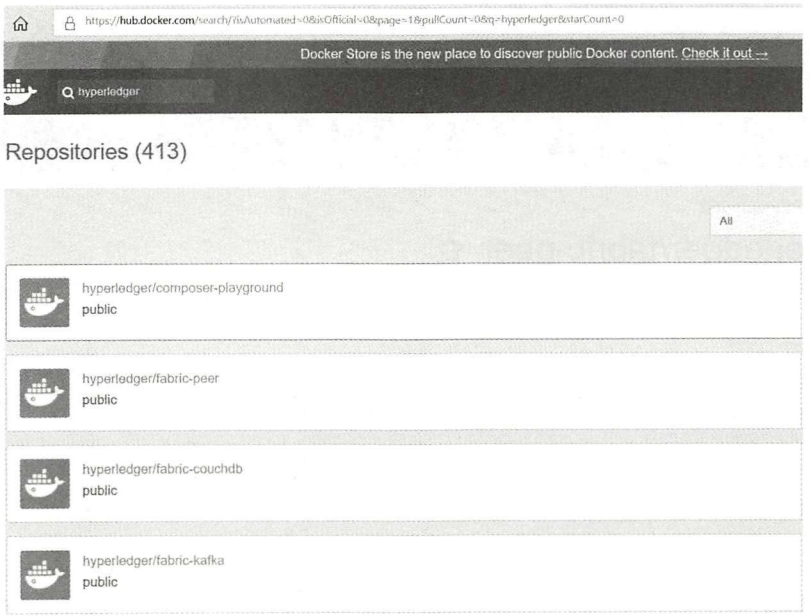


图2–7　检索结果

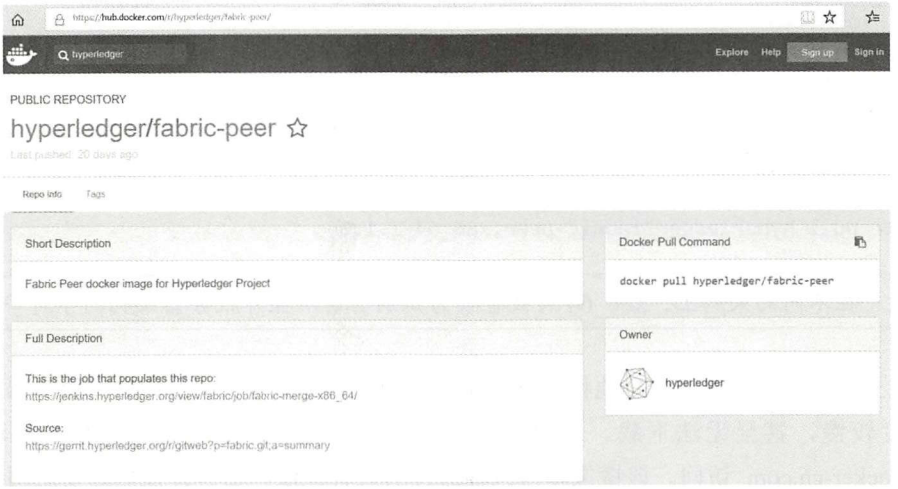


图2–8　下载页面

从图 2-8 中可以找到官方给出的下载方式：

```
docker pull hyperledger/fabric-peer
```

由于在下载 Docker 镜像时如果没有给出指定 tag 会默认使用 lastest，而该方案最终可能导致下载失败，因此在 fabric-peer 下载页选中其 tags 标签，查看当前 fabric-peer 最新版本号，如

图 2-9 所示。

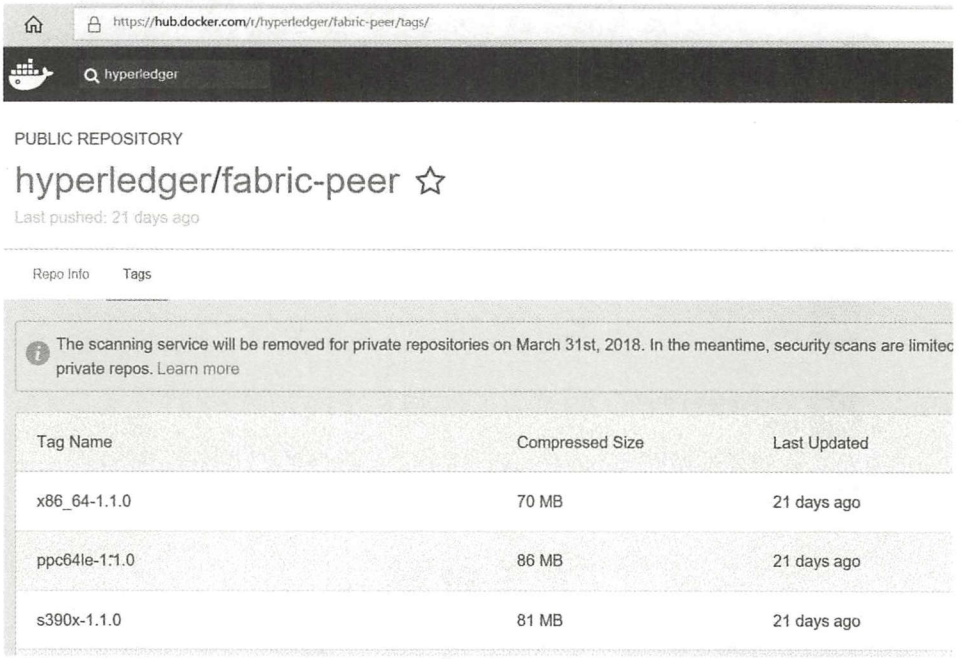


图2-9　版本选择

根据所使用的操作系统情况，选择 x86_64-1.1.0 版本，故最终执行的 Docker 下载命令如下：

```
docker pull hyperledger/fabric-peer:x86_64-1.1.0
```

**注意**：在不同的架构上，x86_64 将被替换为识别当前所操作服务器架构的字符串。

通过上述命令，就可以成功地将 fabric-peer 下载到本地服务器。其中有时服务器可能获取资源速度较慢，甚至无法下载。这里可以使用 Docker 中国官方镜像加速服务，可通过 registry.docker-cn.com 访问。该镜像库只包含流行的公有镜像。私有镜像仍需要从国外镜像库中拉取。

如果遇到上述情况，可以使用如下命令：

```
docker pull registry.docker-cn.com/hyperledger/fabric-peer:x86_64-1.1.0
```

接下来，需要通过上述方案将 Fabric 运行所需镜像全部拉取到本地进行部署，而 Fabric/Peer 运行所需镜像如下：

```
hyperledger/fabric-tools
hyperledger/fabric-peer
hyperledger/fabric-couchdb
hyperledger/fabric-ca
hyperledger/fabric-ccenv
hyperledger/fabric-baseos
```

fabric-tools 是本地客户端镜像，主要用来执行 Peer 节点中的相关操作，如频道、智能合约等。

fabric-peer 是 Fabric 中网络节点镜像，从 1.0 版本开始，Fabric 的 Peer 不再提交数据，全部由客户端完成，但 Peer 依然会对数据进行提交前的验证和背书。

fabric-couchdb 是第三方可插拔式数据库镜像，fabric-couchdb 非必须，如果不使用 CouchDB，则会默认使用 LevelDB，相比较而言使用 CouchDB 功能性更强一些。

fabric-ca 是服务器本地 CA Server，可以执行 fabric-ca-client 中的相关操作，实现登陆、注册及吊销等方法。

当然，Fabric 整个网络的运行不能缺少排序服务，Fabric1.0 之后新增了背书机制，客户端在通过节点背书及合约等一系列验证后，将交易请求发送至排序服务器，排序服务在接收到客户端提交的交易后，会进行数据排序并广播同步至各个 Peer 节点，Fabric/Orderer 运行所需镜像如下：

```
hyperledger/fabric-orderer
```

如果在使用中已经有了部署生产环境的计划，则需要将共识由 Solo 改为 Kafka，而 Kafka 集群部署所需要的镜像如下：

```
hyperledger/fabric-kafka
hyperledger/fabric-zookeeper
```

通过上述步骤，可以将这些必要的镜像由 Docker 服务全部下载至本地，并最终使用 Docker Compose 来启动对应的镜像服务。

如果是在公司内网环境下部署，在没有外网访问权限的条件下，可以事先在可以连接公网访问的服务器上将镜像下载至本地，后续可以将镜像打包并在内网环境的服务器上进行恢复，具体操作将在后续讲述。

为了方便配置 Docker Compose，将所有的镜像 tag 都改为 latest，执行如下格式的命令：

```
docker tag IMAGEID(镜像ID) REPOSITORY:TAG（仓库：标签）
```

例如：

```
docker tag 0403fd1c72c7 docker.io/hyperledger/fabric-tools:latest
```

镜像 ID 可以通过 docker images 命令查看，如执行下述命令将看到如图 2-10 所示结果，图 2-10 也是所有的镜像文件及版本号修改完成后的视图。

```
docker images
```

```
[root@VM_179_237_centos ~]# docker images
REPOSITORY                      TAG                 IMAGE ID            CREATED             SIZE
hyperledger/fabric-tools        latest              0403fd1c72c7        3 weeks ago         1.32GB
hyperledger/fabric-zookeeper    latest              e545dbf1c6af        3 weeks ago         1.31GB
hyperledger/fabric-orderer      latest              e317ca5638ba        3 weeks ago         179MB
hyperledger/fabric-peer         latest              6830dcd7b9b5        3 weeks ago         182MB
hyperledger/fabric-couchdb      latest              b37a08f8a0cb        6 weeks ago         1.48GB
hyperledger/fabric-kafka        latest              dbb5796d915f        6 weeks ago         1.3GB
hyperledger/fabric-javaenv      latest              7cbe6aca3956        6 weeks ago         1.42GB
hyperledger/fabric-ca           latest              cea779a46670        6 weeks ago         238MB
hyperledger/fabric-ccenv        latest              13ed472038d2        6 weeks ago         1.29GB
hyperledger/fabric-baseimage    latest              9f2e9ec7c527        2 months ago        1.27GB
hyperledger/fabric-baseos       latest              4b0cab202084        2 months ago        157MB
hyperledger/fabric-membersrvc   latest              b3654d32e4f9        9 months ago        1.42GB
```

图2-10　镜像

如果下载下来的镜像有问题，可以执行如下命令删除指定 Image ID 的镜像：

```
docker rmi <image id>
```

强制删除命令如下：

```
docker rmi -f<image id>
```

删除全部镜像命令如下：

```
docker rmi $(docker images -q)
```

强制删除全部镜像命令如下：

```
docker rmi -f $(docker images -q)
```

### 2.4.3　镜像备份和迁移

上述第 2.4.2 节中 HyperLedger Fabric 镜像数量较多且容量需求大，一套基本的服务镜像可达 10GB 左右，如果在多台服务器上部署，会花费很多时间。因此，对于上述已经下载至本地的镜像，需要使用 docker save 命令来备份，并通过 scp 命令来将这些镜像文件复制至其他服务器。

该方案也适合将已下载好的镜像部署在无公网访问权限的服务器中。

以镜像 hyperledger/fabric-peer 为例，在执行备份之前，需要先查询当前镜像包的 Image ID，执行命令如下：

```
docker images
```

得到如下结果，可以看到当前已经下载的 fabric-peer 的 Image ID 是 b023f9be0771。

```
hyperledger/fabric-peer  latest  b023f9be0771  2 weeks ago  187MB
```

执行如下命令在/tmp 目录下来生成该镜像的 tar 包：

```
docker save b023f9be0771> /data/fabric-images-1.1.0-release /peer.tar
```

上述命令结构为 docker save IMAGEID(镜像 id) >（文件路径及文件名）。

根据上述命令，对其他已经下载下来的 Fabric 镜像分别执行打包操作，最后在/data/fabric-images-1.1.0-release 目录下执行 ll 查看当前目录镜像文件。正常情况下会看到如图 2-11 所示结果

```
[root@VM_139_63_centos fabric-images-1.1.0-release]# ll
total 3629880
-rw-r--r-- 1 root root  158169600 Mar 21 14:05 fabric-baseos_x86_64-0.4.6.tar
-rw-r--r-- 1 root root  252653568 Mar 21 13:59 fabric-ca_x86_64-1.1.0.tar
-rw-r--r-- 1 root root 1426936320 Mar 21 14:04 fabric-ccenv_x86_64-1.1.0.tar
-rw-r--r-- 1 root root  187258368 Mar 21 14:02 fabric-orderer_x86_64-1.1.0.tar
-rw-r--r-- 1 root root  193645056 Mar 21 14:03 fabric-peer_x86_64-1.1.0.tar
-rw-r--r-- 1 root root 1494656000 Mar 21 14:00 fabric-tools_x86_64-1.1.0.tar
```

图2-11　打包镜像

该视图只截取了一部分，且该视图仅 ll 命令结果用作参考。通过 FTP 可以看到修改过镜像版本名称后的目录大致如图 2-12 所示。

/data/fabric-images-1.1.0-release

| 名称 | 大小 | 类型 | 修改时间 | 属性 | 所有者 |
|---|---|---|---|---|---|
| .. | | | | | |
| fabric-baseos_x86_64-0.4.6.tar | 150.84MB | WinRA... | 2018/3/21, 14:05 | -rw-r--r-- | root |
| fabric-ca_x86_64-1.1.0.tar | 240.95MB | WinRA... | 2018/3/21, 13:59 | -rw-r--r-- | root |
| fabric-ccenv_x86_64-1.1.0.tar | 1.33GB | WinRA... | 2018/3/21, 14:04 | -rw-r--r-- | root |
| fabric-orderer_x86_64-1.1.0.tar | 178.58MB | WinRA... | 2018/3/21, 14:02 | -rw-r--r-- | root |
| fabric-peer_x86_64-1.1.0.tar | 184.67MB | WinRA... | 2018/3/21, 14:03 | -rw-r--r-- | root |
| fabric-tools_x86_64-1.1.0.tar | 1.39GB | WinRA... | 2018/3/21, 14:00 | -rw-r--r-- | root |

图2-12　镜像打包目录

当所有的镜像文件都被打包后，可以通过如下命令格式发送镜像：

```
scp fabric-peer.tar root@10.111.171.217:/tmp/docker/fabric-images
```

通过上述命令可以将 Fabric 镜像文件远程发送至 10.111.171.217 远端服务器本地/tmp/docker/fabric-images 目录下。这里是内网 IP，如果是在内网的环境下，传输速度会很快，外网则根据服务器自身网络情况而定。

对于离线/内网环境，建议通过 FTP 上传下载的手工方式传输，另外，内网传输速度不尽

人意的话，也建议用手工复制。

当远端服务器接收到所有的镜像文件后，可执行如下命令来加载这些镜像文件：

```
docker load < /tmp/docker/fabric-peer.tar
```

镜像加载完成后，可根据第 2.4.2 节最后所述的操作方案，将镜像 tag 改为 lastest，以便配置 Docker Compose。

## 2.5 本章小结

通过本章的讲述，已经了解到如何通过在线或离线的方式获取 HyperLedger Fabric 的源码和必备镜像，以及每一种类型节点所需的镜像类型和含义。

在上手操作的过程中，务必要确保下载的 HyperLedger Fabric 版本与镜像版本相对应，HyperLedger Fabric 的 Zookeeper 和 Kafka 镜像绝大部分可以向下或向上兼容，Peer 和 Orderer 节点所需的必备镜像就不具备强兼容性，需要在部署的时候注意版本的一致性，否则很容器导致执行失败。且 HyperLedger Fabric 的日志报错有时原因并不直白明显，笔者接触的很多朋友有很大一部分都是版本不一致导致一个星期甚至一个月都无法跑通，直到发现镜像版本不一致才通过重新部署解决。

# 第 3 章　End-2-End 案例

End-2-End 是 HyperLedger Fabric 源码中提供的一个 Demo，在 1.0 版本中，它运行的是 Solo 共识，需要一个排序服务和两个组织（每个组织包含两个节点）组网，入门而言较为简单。而从 1.1 版本开始，它改成了 Kafka 共识，利用三个 Zookeeper 和四个 Kafka 组成的两个外围集群，排序服务与节点服务的启动样本与 1.0 版本相同。

与 1.0 版本的 End-2-End 相比，可以说是提高了入门难度，但从另一方面也可以说让入门的学习成本更低。在 1.1 版本的 End-2-End 中，有着更丰富的配置，并由于生产环境的特殊性，根本不会也不可能采用 Solo 方式的共识，让学习入门变得更为直接，避免走不必要的弯路。

但即便如此，作者接触到的绝大多数开发人员都还在 End-2-End 上纠结，很多人能够实现自己自定义配置来启动自定义服务，但依旧没有成功跑通过。

下面，本章将从 1.0 版本开始，将 1.0 和 1.1 版本的 End-2-End 执行一遍，同时将其中的脚本进行一次较为详细的分析，以便更加快速地入门和更加深入地了解 HyperLedger Fabric 整个网络的部署和启动过程。

## 3.1　平台特定文件

HyperLedger Fabric 网络平台部署需要一些特定的二进制文件，如 cryptogen、configtxgen、configtxlator 以及 peer 等。这些二进制文件用于辅助生成证书、密钥以及各项配置文件等。

在第 2.3 节和第 2.4 节中提到过隐私频道、成员安全和共识等内容，这些都需要依托平台特定二进制文件来生成整个网络的所需材料。

也就是说，如果想要成功运行 End-2-End，就必须提前准备好这些文件，而官方也提供了获取该文件的方法。

可以直接参考并按照官方的具体步骤执行，地址为 http://hyperledger-fabric.readthedocs.io/en/latest/ samples.html#download-platform-specific-binaries。

按照官方文档中的描述，需要先执行如下命令：

```
curl -sSL https://goo.gl/6wtTN5 | bash -s 1.1.0
```

上述命令会下载自动化部署脚本，同时也会下载平台特定使用的二进制文件 cryptogen、configtxgen、configtxlator、peer、orderer 以及 fabric-ca-client，把它们放到上述仓库的 bin 目录下。

通常执行上述命令并不能下载，即便是能下载，速度也奇慢无比，故此，这里提供两种离线下载相关二进制文件的方法来代替上述操作。

第一种方法是下载本书发布在网络的资源，下面是平台二进制文件的下载地址。

Platform-specific Binaries for v1.0.0 下载地址为 http://download.csdn.net/download/jiayiyangzhu/10245492

Platform-specific Binaries for v1.1.0 下载地址为 https://download.csdn.net/download/jiayiyangzhu/10330267

另一种方法是分析官方提供的下载方案。

在官网给出的执行命令中并没有给出离线下载地址，且官网也没有离线部署说明，但依然可以在已经下载好的 Fabric 源码中找到下载地址。

在/opt/gopath/src/github.com/hyperledger/fabric/scripts 目录下有一个 bootstrap-1.0.0.sh 脚本文件，如果外网访问条件优越的话，直接运行该脚本即可下载所有所需的 Fabric 镜像文件及官方指定所需的二进制文件。

打开 bootstrap.sh（目前最新版本为 1.1.0，以实际为准），找到其中对“Downloading platform binaries”的输出行，目前所见是“echo "===> Downloading platform binaries"”，查看其指向的下载地址，可以得到一个官网提供的离线下载网址，如下所示：

```
    echo "===> Downloading platform specific fabric binaries"
    curl https://nexus.hyperledger.org/content/repositories/releases/org/hyperledger/
fabric/hyperledger-fabric/${ARCH}-${VERSION}/hyperledger-fabric-${ARCH}-
${VERSION}.tar.gz | tar xz

    echo "===> Downloading platform specific fabric-ca-client binary"
    curl https://nexus.hyperledger.org/content/repositories/releases/org/hyperledger/
fabric-ca/hyperledger-fabric-ca/${ARCH}-${VERSION}/hyperledger-fabric-ca-${ARCH}-
${VERSION}.tar.gz | tar xz
```

根据上下文意思及当前所使用的版本信息，可以得到最终的离线下载文件地址为 https://nexus.hyperledger.org/content/repositories/releases/org/hyperledger/fabric/hyperledger-fabric/linux-amd64-1.1.0/。

直接根据系统版本进行选择并将压缩包下载至本地即可，根据官网的介绍，解压后会得到一个 bin 文件夹。

## 3.2　运行e2e_cli

为了方便测试和了解，将首先运行一遍 1.0 版本的 e2e_cli，它与 1.1 版本的区别在前文已经提到过，这里直接开始进入实操步骤。

因为之前下载至本地的是 Fabric1.1 版本的源码，那么在该版本中的 e2e_cli 也是 1.1 版本的，需要做一些切换的步骤，也就是使用 Fabric1.0 版本中的 e2e_cli 替换 Fabric1.1 版本中的 e2e_cli。

进入/home/docker/github.com/hyperledger/fabric/examples 目录，将 e2e_cli 文件彻底删除，同时将 Fabric1.0 版本的源码下载至本地，并将其中的 e2e_cli 上传至服务器的/home/docker/github.com/hyperledger/fabric/examples 目录中。

在此之前，需要确定本地镜像是否已经下载且准备好 tag，可以参考第 2.4.2 节的步骤进行操作，最终本次运行案例需要的镜像如下：

```
hyperledger/fabric-orderer
hyperledger/fabric-tools
hyperledger/fabric-peer
hyperledger/fabric-couchdb
hyperledger/fabric-ca
hyperledger/fabric-ccenv
hyperledger/fabric-baseos
```

进入到/home/docker/github.com/hyperledger/fabric/examples/e2e_cli 目录下，文件结构如图 3-1 所示。

/home/docker/github.com/hyperledger/fabric/examples/e2e_cli

| 名称 | 大小 | 类型 | 修改时间 | 属性 | 所有者 |
|---|---|---|---|---|---|
| .. | | | | | |
| base | | 文件夹 | 2018/4/6, 16:21 | drwxr-... | root |
| channel-artifacts | | 文件夹 | 2018/4/6, 16:21 | drwxr-... | root |
| examples | | 文件夹 | 2018/4/6, 16:21 | drwxr-... | root |
| scripts | | 文件夹 | 2018/4/6, 16:21 | drwxr-... | root |
| configtx.yaml | 5KB | YAML ... | 2018/4/6, 16:21 | -rw-r--r-- | root |
| crypto-config.yaml | 4KB | YAML ... | 2018/4/6, 16:21 | -rw-r--r-- | root |
| docker-compose-cli.yaml | 3KB | YAML ... | 2018/4/6, 16:21 | -rw-r--r-- | root |
| docker-compose-couch.yaml | 2KB | YAML ... | 2018/4/6, 16:21 | -rw-r--r-- | root |
| docker-compose-e2e-template.y... | 2KB | YAML ... | 2018/4/6, 16:21 | -rw-r--r-- | root |
| docker-compose-e2e.yaml | 3KB | YAML ... | 2018/4/6, 16:21 | -rw-r--r-- | root |
| download-dockerimages.sh | 2KB | SH 文件 | 2018/4/6, 16:21 | -rw-r--r-- | root |
| end-to-end.rst | 41KB | RST 文件 | 2018/4/6, 16:21 | -rw-r--r-- | root |
| generateArtifacts.sh | 4KB | SH 文件 | 2018/4/6, 16:21 | -rw-r--r-- | root |
| network_setup.sh | 3KB | SH 文件 | 2018/4/6, 16:21 | -rw-r--r-- | root |

图3-1　文件结构

该目录下的文件已经被替换成 1.0 版本中的文件，但并不影响脚本的执行。

该目录下的 network_setup.sh 是一件测试脚本，该脚本启动 5 个 Docker 容器，其中 4 个容

器运行 Peer 节点和 1 个容器运行 Orderer 节点，它组成一个 Fabric 网络。另外，还有一个 cli 容器用于执行创建 channel、加入 channel、安装和执行 chaincode 等操作。测试用的 chaincode 定义了两个变量，在实例化的时候会给这两个变量赋予初始值，并通过 invoke 操作可以使两个变量的值发生变化。

通过以下命令执行测试：

```
bash network_setup.sh up
```

中途可能会遇到类似如下所示的提示：

```
/bin/bash: ./scripts/script.sh: Permission denied
```

如提示中描述所示，script.sh 脚本缺少权限，根据提示给 script.sh 对应的读写权限即可。但在执行该操作之前需要先将已启动的服务关闭，因为之前的操作已经将 4 个 Peer 节点、Orderer 节点以及 cli 客户端容器启动，如果不执行关闭服务操作，后续再度运行的时候还会报出其他难以排查原因的错误。

这其中很大的问题在于环境污染，运行 Fabric 的网络需要一个干净的环境，对于与之相关的过往被废弃的容器等组件需要彻底关闭，故此执行如下命令关闭服务：

```
bash network_setup.sh down
```

随后执行如下命令给 script.sh 赋权：

```
sudo chmod +x scripts/script.sh
```

最终得到如图 3-2 所示的结果。

```
Creating peer1.org2.example.com ... done
Creating cli ... done
Creating peer0.org2.example.com ...
Creating peer0.org1.example.com ...
Creating orderer.example.com ...
Creating peer1.org1.example.com ...
Creating cli ...
/bin/bash: ./scripts/script.sh: Permission denied
^Z
[2]+  Stopped                 bash network_setup.sh up
[root@VM_139_63_centos e2e_cli]# bash network_setup.sh down
setting to default channel 'mychannel'
WARNING: The CHANNEL_NAME variable is not set. Defaulting to a blank string.
WARNING: The TIMEOUT variable is not set. Defaulting to a blank string.
Stopping cli ... done
Removing cli                    ... done
Removing peer1.org2.example.com ... done
Removing orderer.example.com    ... done
Removing peer0.org1.example.com ... done
Removing peer0.org2.example.com ... done
Removing peer1.org1.example.com ... done
Removing network e2ecli_default
05e47b096850
---- No images available for deletion ----
[root@VM_139_63_centos e2e_cli]# sudo chmod +x scripts/script.sh
[root@VM_139_63_centos e2e_cli]# ll scripts/script.sh
-rwxr-xr-x 1 root root 8745 Apr  6 16:21 scripts/script.sh
```

图3-2　授权操作

随后，再次执行如下命令启动 e2e_cli：

```
bash network_setup.sh up
```

也许，还会出现如图 3-3 所示错误提示。

```
 ____    _____      _      ____    _____           ____    _____   ____
/ ___|  |_   _|    / \    |  _ \  |_   _|         / ___|  |___ /  |___ \
\___ \    | |     / _ \   | |_) |   | |    _____  | |  _     |_ \    __) |
 ___) |   | |    / ___ \  |  _ <    | |   |_____| | |_| |  ___) |  / __/
|____/    |_|   /_/   \_\ |_| \_\   |_|           \____| |____/  |_____|

Channel name : mychannel
Creating channel...
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
CORE_PEER_LOCALMSPID=Org1MSP
CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt
CORE_PEER_TLS_ENABLED=true
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ID=cli
CORE_LOGGING_LEVEL=DEBUG
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
2018-04-06 08:35:00.682 UTC [main] main -> ERRO 001 Cannot run peer because cannot init crypto, missing /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
org1.example.com/msp folder
!!!!!!!!!!!!!!! Channel creation failed !!!!!!!!!!!!!!!
==================== ERROR !!! FAILED to execute End-2-End Scenario ====================
```

图3-3　启动报错

很多读者会卡在这里，四处寻找答案而始终得不到解决方案。在这需要说明的是，HyperLedger Fabric 的报错有时并非特别友好，并非是它没有给出错误的原因，而是需要开发人员仔细地阅读每一行日志，日志分析往往能带给开发人员想要的答案。

现在继续这个错误提示，继续往前翻看日志，可以看到如图 3-4 所示。

```
############################################################
##### Generate certificates using cryptogen tool #########
############################################################
generateArtifacts.sh: line 58: /home/docker/github.com/hyperledger/fabric/examples/e2e_cli/../../release/linux-amd64/bin/cryptogen: No such file or directory

generateArtifacts.sh: line 33: cd: crypto-config/peerOrganizations/org1.example.com/ca/: No such file or directory
ls: cannot access *_sk: No such file or directory
generateArtifacts.sh: line 37: cd: crypto-config/peerOrganizations/org2.example.com/ca/: No such file or directory
ls: cannot access *_sk: No such file or directory
Building configtxgen
which: no git in (/usr/lib64/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/usr/local/go/bin:/root/bin)
make: Entering directory `/home/docker/github.com/hyperledger/fabric'
Makefile:92: *** "No git in PATH: Check dependencies".  Stop.
make: Leaving directory `/home/docker/github.com/hyperledger/fabric'
############################################################
#########  Generating Orderer Genesis block ##############
############################################################
generateArtifacts.sh: line 78: /home/docker/github.com/hyperledger/fabric/examples/e2e_cli/../../release/linux-amd64/bin/configtxgen: No such file or directory

#################################################################
### Generating channel configuration transaction 'channel.tx' ###
#################################################################
generateArtifacts.sh: line 84: /home/docker/github.com/hyperledger/fabric/examples/e2e_cli/../../release/linux-amd64/bin/configtxgen: No such file or directory

#################################################################
#######    Generating anchor peer update for Org1MSP   ##########
#################################################################
generateArtifacts.sh: line 90: /home/docker/github.com/hyperledger/fabric/examples/e2e_cli/../../release/linux-amd64/bin/configtxgen: No such file or directory
```

图3-4　错误信息

在这段日志中，可以看到很多地方都报出了“No such file or directory”提示，而第一个报出这样提示的位置根据常理，应该就是错误的根源。

可以看到第一处报出该提示的日志如下所示：

```
    ############################################################
    ##### Generate certificates using cryptogen tool #########
    ############################################################
    generateArtifacts.sh: line 58: /home/docker/github.com/hyperledger/fabric/
examples/e2e_cli/../../release/linux-amd64/bin/cryptogen: No such file or directory
```

也就是说，cryptogen 文件并不在/home/docker/github.com/hyperledger/fabric/examples/e2e_cli/../../release/linux-amd64/bin/目录下。

通过第 3.1 节的阅读，大致已经可以分析出，这里是缺少了官方平台特定的二进制文件而导致报错。根据提示所指，将之前下载的官方平台特定的二进制文件上传至报错指向的目录中，最终目录结果如图 3-5 所示。

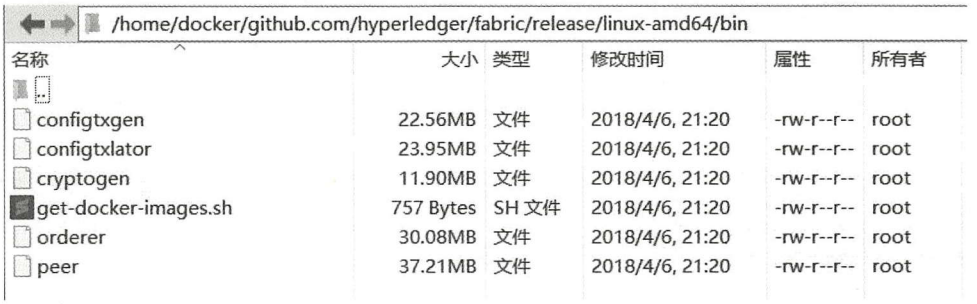


图3-5 目录结构

上传完成后，需要执行如下命令关闭已经启动的服务：

```
bash network_setup.sh down
```

随后执行如下命令再次启动服务：

```
bash network_setup.sh up
```

这个案例可能让开发人员卡住很久，因为作者之前发表过《Hyperledger Fabric 1.0 从零开始》系列博文，其中在描述该案例的时候只是将具体的操作步骤和成功结果截图并告知出来，而其中的坑也是通过其他方式告知。所以，在编写本书的时候，会按照一个正常的首次接触该平台的开发人员行动方式来处理其中的问题，并把坑一个一个地走一遍。

当上传操作后，在执行该案例的时候会有权限要求，但最终的报错信息依旧如图 3-4 所示，需要通过对日志进行向前追述来获取真正的原因。通过查阅，可以得到最终的错误分析如图 3-6 所示。

```
############################################################
##### Generate certificates using cryptogen tool #########
############################################################
generateArtifacts.sh: line 58: /home/docker/github.com/hyperledger/fabric/examples/e2e_cli/../../release/linux-amd64/bin/cryptogen: Permission denied

generateArtifacts.sh: line 33: cd: crypto-config/peerOrganizations/org1.example.com/ca/: No such file or directory
ls: cannot access *_sk: No such file or directory
generateArtifacts.sh: line 37: cd: crypto-config/peerOrganizations/org2.example.com/ca/: No such file or directory
ls: cannot access *_sk: No such file or directory
Using configtxgen -> /home/docker/github.com/hyperledger/fabric/examples/e2e_cli/../../release/linux-amd64/bin/configtxgen
############################################################
#########  Generating Orderer Genesis block ##############
############################################################
generateArtifacts.sh: line 78: /home/docker/github.com/hyperledger/fabric/examples/e2e_cli/../../release/linux-amd64/bin/configtxgen: Permission denied

#############################################################
### Generating channel configuration transaction 'channel.tx' ###
#############################################################
generateArtifacts.sh: line 84: /home/docker/github.com/hyperledger/fabric/examples/e2e_cli/../../release/linux-amd64/bin/configtxgen: Permission denied

#################################################################
#######    Generating anchor peer update for Org1MSP   ##########
#################################################################
generateArtifacts.sh: line 90: /home/docker/github.com/hyperledger/fabric/examples/e2e_cli/../../release/linux-amd64/bin/configtxgen: Permission denied

#################################################################
#######    Generating anchor peer update for Org2MSP   ##########
#################################################################
generateArtifacts.sh: line 96: /home/docker/github.com/hyperledger/fabric/examples/e2e_cli/../../release/linux-amd64/bin/configtxgen: Permission denied
```

图3-6 错误日志

其中关键信息表述如下：

```
    ############################################################
    ##### Generate certificates using cryptogen tool #########
    ############################################################
    generateArtifacts.sh: line 58: /home/docker/github.com/hyperledger/fabric/
examples/e2e_cli/../../release/linux-amd64/bin/cryptogen: Permission denied
```

可以看出官方平台特定的二进制文件 cryptogen 已经成功加载，但因为没有权限，从而导致访问被拒，与之前处理 script.sh 的方式相同，给 cryptogen 以及 configtxgen 都进行赋权操作。

因为 configtxgen 也会在本次运行的案例中生成相关配置文件，故此也需要进行一次赋权操作，这里在后面的章节中会有具体的说明。

这里进入官方平台特定的二进制文件目录并执行授权操作，命令如下所示：

```
cd ../../release/linux-amd64/bin/
sudo chmod +x cryptogen
sudo chmod +x configtxgen
```

最终可以看到如图 3-7 所示结果。

```
[root@VM_139_63_centos e2e_cli]# cd ../../release/linux-amd64/bin/
[root@VM_139_63_centos bin]# ll
total 128872
-rw-r--r-- 1 root root 23653336 Apr  6 21:20 configtxgen
-rw-r--r-- 1 root root 25113376 Apr  6 21:20 configtxlator
-rw-r--r-- 1 root root 12473976 Apr  6 21:20 cryptogen
-rw-r--r-- 1 root root      757 Apr  6 21:20 get-docker-images.sh
-rw-r--r-- 1 root root 31536304 Apr  6 21:20 orderer
-rw-r--r-- 1 root root 39016824 Apr  6 21:20 peer
[root@VM_139_63_centos bin]# sudo chmod +x cryptogen
[root@VM_139_63_centos bin]# sudo chmod +x configtxgen
[root@VM_139_63_centos bin]# ll
total 128872
-rwxr-xr-x 1 root root 23653336 Apr  6 21:20 configtxgen
-rw-r--r-- 1 root root 25113376 Apr  6 21:20 configtxlator
-rwxr-xr-x 1 root root 12473976 Apr  6 21:20 cryptogen
-rw-r--r-- 1 root root      757 Apr  6 21:20 get-docker-images.sh
-rw-r--r-- 1 root root 31536304 Apr  6 21:20 orderer
-rw-r--r-- 1 root root 39016824 Apr  6 21:20 peer
```

图3-7　授权结果

授权操作后，回到/home/docker/github.com/hyperledger/fabric/examples/e2e_cli 目录下，再次执行如下命令，关闭已经启动的服务：

```
bash network_setup.sh down
```

随后执行如下命令再次启动服务：

```
bash network_setup.sh up
```

解决前面的几个报错问题后，已经基本扫平了所有有关 e2e_cli 案例执行的障碍，如果一切顺利的话，将会出现如图 3-8 所示结果。

```
2018-04-06 13:45:04.540 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-04-06 13:45:04.541 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-04-06 13:45:04.541 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-04-06 13:45:04.541 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-04-06 13:45:04.541 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java chaincode disabled
2018-04-06 13:45:04.541 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AAB070A6708031A0C08E0F39DD60510...6D7963631A0A0A0571756572790A0161
2018-04-06 13:45:04.541 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: 1C8D9549DA5BA16D5A16BE1EFB180828C4503637099357A1C67A58D68FD06013
Query Result: 90
2018-04-06 13:45:35.179 UTC [main] main -> INFO 008 Exiting.....
===================== Query on PEER3 on channel 'mychannel' is successful =====================

===================== All GOOD, End-2-End execution completed =====================
```

图3-8　运行成功视图

至此，已经完全跑通了 e2e_cli 在 Fabric1.0 版本中的案例，接下来开始再跑一遍 Fabric1.1 版本的案例。

执行如下命令查看已经启动的容器有哪些：

```
docker ps
```

结果如图 3-9 所示。

```
[root@VM_139_63_centos e2e_cli]# docker ps
CONTAINER ID        IMAGE                                                
```

## 3.3 e2e_cli案例分析

### 3.3.1 容器服务脚本

e2e_cli 项目的入口位于 network_setup.sh，在启动的时候给出的传入参数是 up，其实仔细阅读该脚本文件，内置允许传入 4 个参数，分别是 UP_DOWN（执行类型）、CH_NAME（频道名称）、CLI_TIMEOUT（客户端超时设置）和 IF_COUCHDB（是否启动 CouchDB 版本 yaml 文件）。

network_setup.sh 脚本的具体内容如下所示：

```
#!/bin/bash
#
# Copyright IBM Corp. All Rights Reserved.
#
# SPDX-License-Identifier: Apache-2.0
#


UP_DOWN="$1"
CH_NAME="$2"
CLI_TIMEOUT="$3"
IF_COUCHDB="$4"

: ${CLI_TIMEOUT:="10000"}

COMPOSE_FILE=docker-compose-cli.yaml
COMPOSE_FILE_COUCH=docker-compose-couch.yaml
#COMPOSE_FILE=docker-compose-e2e.yaml

function printHelp () {
 echo "Usage: ./network_setup <up|down> <\$channel-name> <\$cli_timeout>
<couchdb>.\nThe arguments must be in order."
}

function validateArgs () {
 if [ -z "${UP_DOWN}" ]; then
      echo "Option up / down / restart not mentioned"
      printHelp
      exit 1
 fi
 if [ -z "${CH_NAME}" ]; then
      echo "setting to default channel 'mychannel'"
      CH_NAME=mychannel
```

```
    fi
  }

  function clearContainers () {
        CONTAINER_IDS=$(docker ps -aq)
        if [ -z "$CONTAINER_IDS" -o "$CONTAINER_IDS" = " " ]; then
              echo "---- No containers available for deletion ----"
        else
              docker rm -f $CONTAINER_IDS
        fi
  }

  function removeUnwantedImages() {
        DOCKER_IMAGE_IDS=$(docker images | grep "dev\|none\|test-vp\|peer[0-9]-" |
awk '{print $3}')
        if [ -z "$DOCKER_IMAGE_IDS" -o "$DOCKER_IMAGE_IDS" = " " ]; then
              echo "---- No images available for deletion ----"
        else
              docker rmi -f $DOCKER_IMAGE_IDS
        fi
  }

  function networkUp () {
     if [ -f "./crypto-config" ]; then
       echo "crypto-config directory already exists."
     else
       #Generate all the artifacts that includes org certs, orderer genesis block,
       # channel configuration transaction
       source generateArtifacts.sh $CH_NAME
     fi

     if [ "${IF_COUCHDB}" == "couchdb" ]; then
       CHANNEL_NAME=$CH_NAME TIMEOUT=$CLI_TIMEOUT docker-compose -f $COMPOSE_FILE -f
$COMPOSE_FILE_COUCH up -d 2>&1
     else
       CHANNEL_NAME=$CH_NAME TIMEOUT=$CLI_TIMEOUT docker-compose -f $COMPOSE_FILE up
-d 2>&1
     fi
     if [ $? -ne 0 ]; then
   echo "ERROR !!!! Unable to pull the images "
   exit 1
     fi
     docker logs -f cli
  }
```

```
function networkDown () {
   docker-compose -f $COMPOSE_FILE down

   #Cleanup the chaincode containers
   clearContainers

   #Cleanup images
   removeUnwantedImages

   # remove orderer block and other channel configuration transactions and certs
   rm -rf channel-artifacts/*.block channel-artifacts/*.tx crypto-config
}

validateArgs

#Create the network using docker compose
if [ "${UP_DOWN}" == "up" ]; then
 networkUp
elif [ "${UP_DOWN}" == "down" ]; then ## Clear the network
 networkDown
elif [ "${UP_DOWN}" == "restart" ]; then ## Restart the network
 networkDown
 networkUp
else
 printHelp
 exit 1
fi
```

通过阅读，在该脚本入口处如果没有指定 CLI_TIMEOUT，则默认为 10000。另外，也指定了两个已经准备好的官方 yaml 可执行文件，如下所示：

```
COMPOSE_FILE=docker-compose-cli.yaml
COMPOSE_FILE_COUCH=docker-compose-couch.yaml
```

第 3.2 节中因为没有指定 IF_COUCHDB，所以默认执行的是 docker-compose-cli.yaml，该文件与 network_setup.sh 在同级目录下。

当执行 network_setup.sh up 时会默认执行其内部的 networkUp 方法，按照脚本的意思是首先判断是否存在 crypto-config 目录，默认情况下是不存在的。这点通过 networkDown 方法也可以获知，在执行关闭服务操作的时候，会将该目录及目录下的所有文件内容整体删除。

在判断出没有 crypto-config 目录后，会调用 generateArtifacts.sh 脚本创建 crypto-config 目录及所需的区块链网络证书等文件。

这时需要进一步确定该操作在 generateArtifacts.sh 做了什么，而通过对 generateArtifacts.sh 的深入了解，也能够清楚官方平台特定二进制文件的用途和基本用法。

在 generateArtifacts.sh 脚本开头，有一步需要关注的操作如下所示：

```
export FABRIC_ROOT=$PWD/../..
export FABRIC_CFG_PATH=$PWD
```

这里指定了 Fabric 的根目录在 Fabric 目录下，且配置文件路径就在当前路径下。这一步操作是灵活的，但一般情况下，当前目录就是项目的工作目录，因为绝大部分操作都会在当前目录下进行。

随后，该脚本会判断当前操作系统的版本和类型，具体如下命令所示：

```
OS_ARCH=$(echo "$(uname -s|tr '[:upper:]' '[:lower:]'|sed 's/mingw64_nt. */windows
/')-$(uname -m | sed 's/x86_64/amd64/g')" | awk '{print tolower($0)}')
```

该脚本中有三个可执行方法，默认执行顺序是 generateCerts、replacePrivateKey 和 generateChannelArtifacts。

首先看下 generateCerts 方法，具体如下所示：

```
function generateCerts (){
 CRYPTOGEN=$FABRIC_ROOT/release/$OS_ARCH/bin/cryptogen

 if [ -f "$CRYPTOGEN" ]; then
          echo "Using cryptogen -> $CRYPTOGEN"
 else
    echo "Building cryptogen"
    make -C $FABRIC_ROOT release
 fi

 echo
 echo "##########################################################"
 echo "##### Generate certificates using cryptogen tool #########"
 echo "##########################################################"
 $CRYPTOGEN generate --config=./crypto-config.yaml
 echo
}
```

在该方法中，首先指定了 cryptogen 文件的路径，以便后续执行生成组织及节点文件信息。这一步操作也是解决第 3.2 节中如图 3-4 中的错误，也解释了为什么需要提前准备好官方平台特定二进制文件的原因。

在该方法中，还会执行如下命令进行生成操作：

```
$CRYPTOGEN generate --config=./crypto-config.yaml
```

如上述命令中所指出，对应的配置文件就是当前目录下的 crypto-config.yaml。

crypto-config.yaml 的具体内容如下所示：

```
# Copyright IBM Corp. All Rights Reserved.
#
# SPDX-License-Identifier: Apache-2.0
#

# ---------------------------------------------------------------------------
# "OrdererOrgs" - Definition of organizations managing orderer nodes
# ---------------------------------------------------------------------------
OrdererOrgs:
  # ---------------------------------------------------------------------------
  # Orderer
  # ---------------------------------------------------------------------------
  - Name: Orderer
    Domain: example.com
    # ---------------------------------------------------------------------------
    # "Specs" - See PeerOrgs below for complete description
    # ---------------------------------------------------------------------------
    Specs:
      - Hostname: orderer
# ---------------------------------------------------------------------------
# "PeerOrgs" - Definition of organizations managing peer nodes
# ---------------------------------------------------------------------------
PeerOrgs:
  # ---------------------------------------------------------------------------
  # Org1
  # ---------------------------------------------------------------------------
  - Name: Org1
    Domain: org1.example.com
    # ---------------------------------------------------------------------------
    # "Specs"
    # ---------------------------------------------------------------------------
    # Uncomment this section to enable the explicit definition of hosts in your
    # configuration.  Most users will want to use Template, below
    #
    # Specs is an array of Spec entries.  Each Spec entry consists of two fields:
    #   - Hostname:   (Required) The desired hostname, sans the domain.
    #   - CommonName: (Optional) Specifies the template or explicit override for
    #                 the CN.  By default, this is the template:
    #
    #                            "{{.Hostname}}.{{.Domain}}"
    #
```

```
      #              which obtains its values from the Spec.Hostname and
      #              Org.Domain, respectively.
      # ---------------------------------------------------------------------------
      # Specs:
      #   - Hostname: foo # implicitly "foo.org1.example.com"
      #     CommonName: foo27.org5.example.com # overrides Hostname-based FQDN set
above
      #   - Hostname: bar
      #   - Hostname: baz
      # ---------------------------------------------------------------------------
      # "Template"
      # ---------------------------------------------------------------------------
      # Allows for the definition of 1 or more hosts that are created sequentially
      # from a template. By default, this looks like "peer%d" from 0 to Count-1.
      # You may override the number of nodes (Count), the starting index (Start)
      # or the template used to construct the name (Hostname).
      #
      # Note: Template and Specs are not mutually exclusive.  You may define both
      # sections and the aggregate nodes will be created for you.  Take care with
      # name collisions
      # ---------------------------------------------------------------------------
      Template:
        Count: 2
        # Start: 5
        # Hostname: {{.Prefix}}{{.Index}} # default
      # ---------------------------------------------------------------------------
      # "Users"
      # ---------------------------------------------------------------------------
      # Count: The number of user accounts _in addition_ to Admin
      # ---------------------------------------------------------------------------
      Users:
        Count: 1
    # ---------------------------------------------------------------------------
    # Org2: See "Org1" for full specification
    # ---------------------------------------------------------------------------
    - Name: Org2
      Domain: org2.example.com
      Template:
        Count: 2
      Users:
        Count: 1
```

在 crypto-config.yaml 配置文件中有两大关键的定义如下：

1．OrdererOrgs：排序节点的组织的定义

在定义排序节点的时候，会同时定义节点名称、节点域名以及规范列表中节点所需的不带域的主机名。

2．PeerOrgs：Peer 节点的组织的定义

在定义 Peer 节点时，与排序节点一样也有类似的配置，包括节点名称和节点域名。但 Peer 节点的配置有两种方案，一种是通过规范列表来指定，另一种是通过模板创建的方式实现。

上述关键 1 中的配置可以定义多个排序服务节点，并以此来实现排序服务的集群部署，具体部署方案会在后面的章节中详细介绍。

根据上述关键 2 中 Peer 节点的组织的定义里的描述，演练两种写法，第一种是官方配置文件中的写法，如下：

```
- Name: Org1
    Domain: org1.example.com
    Template:
      Count: 2
    Users:
      Count: 1
```

在该配置文件中有指定组织名称、域名和模板信息，在模板信息 Template 中指定了 Count 数量，是指允许通过该模板定义创建 0 到 Count-1 个所属组织节点或主机。而 Users 则是指除了 Admin 之外的创建用户，其下 Count 定义代表创建的用户数量。

第二种是 Peer 节点的组织定义配置，写法如下：

```
- Name: Org1
    Domain: org1.example.com
    Specs:
      - Hostname: foo
        CommonName: foo27.org5.example.com
      - Hostname: bar
      - Hostname: baz
```

与第一种官方默认写法略有区别的是没有采用 Template 模板配置方案。在第二种写法配置中 Specs 是指一组规范条目，其中每个规范条目由两个字段组成。

如上所述，这两个字段分别是 Hostname 和 CommonName。

其中，Hostname 字段的含义与定义排序节点的时候同时定义节点所需的不带域的主机名一样。CommonName 是一个可选参数，可以通过显示的重写来指定配置模板文件，如上述配置文件中 foo 指定了节点文件目录和域名为 foo27.org5.example.com。而如 bar 和 baz 并未指定，它的默认格式是"{{.Hostname}}.{{.Domain}}"，即最终 bar 和 baz 所生成的节点文件目录

和域名分别是 bar.org1.example.com 和 baz.org1.example.com。

相对于第二种显示指定的方式，在默认组织机构信息和节点需求较大的情况下，选择第一种明显是更好的方式，而一般情况下选择第一种即可。

且第一种 Template 模板配置方案与第二种 Specs 指定显示规范并不冲突，可以在编写 crypto-config.yaml 配置文件的时候同时定义这两种方案，并创建聚合节点，但需要注意其中的名称不要冲突。

通过简单理解了 crypto-config.yaml 配置文件之后，再回过头看 generateArtifacts.sh 中的 $CRYPTOGEN generate --config=./crypto-config.yaml 命令，就比较容易理解。该操作就是需要通过 crypto-config.yaml 来生成排序及验收或同步节点证书及文件等内容，最终会生成在当前目录下的 crypto-config 目录中，如图 3-11 所示。

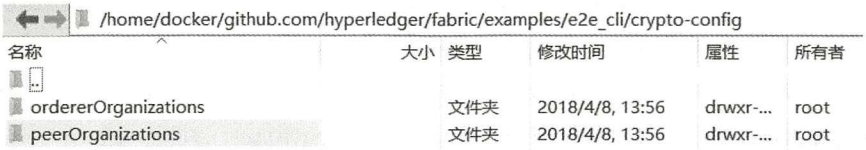


图3-11 crypto-config目录

当 generateArtifacts.sh 脚本中 generateCerts 方法执行完成并生成对应排序及验证或同步节点证书等配置文件后，会继续执行 replacePrivateKey 方法。

replacePrivateKey 方法具体如下：

```
function replacePrivateKey () {
 ARCH=`uname -s | grep Darwin`
 if [ "$ARCH" == "Darwin" ]; then
     OPTS="-it"
 else
     OPTS="-i"
 fi

 cp docker-compose-e2e-template.yaml docker-compose-e2e.yaml

       CURRENT_DIR=$PWD
       cd crypto-config/peerOrganizations/org1.example.com/ca/
       PRIV_KEY=$(ls *_sk)
       cd $CURRENT_DIR
       sed $OPTS "s/CA1_PRIVATE_KEY/${PRIV_KEY}/g" docker-compose-e2e.yaml
       cd crypto-config/peerOrganizations/org2.example.com/ca/
       PRIV_KEY=$(ls *_sk)
       cd $CURRENT_DIR
```

```
        sed $OPTS "s/CA2_PRIVATE_KEY/${PRIV_KEY}/g" docker-compose-e2e.yaml
    }
```

在 replacePrivateKey 方法中，会替换 docker-compose-e2e-template.yaml 文件中的 CA1_PRIVATE_KEY 为当前目录 crypto-config/peerOrganizations/org1.example.com/ca/下的以“_sk”结尾的文件名，同时替换 CA2_PRIVATE_KEY 为当前目录 crypto-config/peerOrganizations/org2.example.com/ca/下的以“_sk”结尾的文件名。

最终生成的新文件被创建在当前文件夹下并命名为 docker-compose-e2e.yaml，在该文件中定义了 CA 的 CERTFILE 及 KEYFILE，同时也通过 command 内置参数显示启动了 Fabric-CA 服务端。

最后，在 generateArtifacts.sh 脚本中执行 generateChannelArtifacts 方法，其中设置 configtxgen 文件路径与 generateCerts 方法一致，都在方法开始的时候进行指定，以便后续执行生成创世区块、频道文件等信息。

在该方法中会执行如下命令列表进行生成操作：

```
    $CONFIGTXGEN -profile TwoOrgsOrdererGenesis -outputBlock ./channel-
artifacts/genesis.block
    $CONFIGTXGEN -profile TwoOrgsChannel -outputCreateChannelTx ./channel-
artifacts/channel.tx -channelID $CHANNEL_NAME
    $CONFIGTXGEN -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-
artifacts/Org1MSPanchors.tx -channelID $CHANNEL_NAME -asOrg Org1MSP
    $CONFIGTXGEN -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-
artifacts/Org2MSPanchors.tx -channelID $CHANNEL_NAME -asOrg Org2MSP
```

如上述命令中所指出，对应的配置文件就是当前目录下的 configtx.yaml。

configtx.yaml 中主要有如下 5 大配置信息。

### 1. Profiles：配置文件

不同的配置概要文件可以在这里进行编码，以指定为 configtxgen 工具的参数。

### 2. Organizations：组织节点信息

定义了不同的组织标志，这些标志将在稍后的配置中被引用。

### 3. Orderer：启动模式

定义了将编码转换为配置事务的值，或者用于 orderer 相关参数的创世纪块。

4. Application：应用

定义了将编码转换为配置事务的值，或者用于应用程序相关参数的创世纪块。

5. Capabilities：功能特性集合

这个配置是 Fabric1.1 版本新推出的概念，用于定义 Fabric 网络的功能。故此，该配置属性不能与 Fabric1.0.x 的 Peer 节点及排序服务节点在 Fabric 网络中混合使用。该功能定义了必须存在于 Fabric 二进制文件中的特性，以便该二进制文件安全地参与到 Fabric 网络中。例如：如果计划新增一个新的 MSP 类型，新的二进制文件可能会识别并验证这种类型的签名，而没有使用这种支持的旧版二进制文件将无法验证这些交易。这就有可能会导致不同版本的 Fabric 二进制文件有不同的 World States。反过来也一样，定义频道的操作会通知二进制文件，如果没有这项功能，则 Peer 节点等会停止处理事务，直到被升级到 Fabric1.1。对于 Fabric1.0.x 来说，如果定义了任何该功能参数（包括有功能关闭的映射），都会导致 Fabric1.0.x 版本的 Peer 节点崩溃丢失。

接下来，开始对上述几个配置信息进行简要说明。

首先是 Profiles 配置，具体示例如下所示：

```
Profiles:

    TwoOrgsOrdererGenesis:
        Capabilities:
            <<: *ChannelCapabilities
        Orderer:
            <<: *OrdererDefaults
            Organizations:
                - *OrdererOrg
            Capabilities:
                <<: *OrdererCapabilities
        Consortiums:
            SampleConsortium:
                Organizations:
                    - *Org1
                    - *Org2
    TwoOrgsChannel:
        Consortium: SampleConsortium
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Org1
                - *Org2
```

```
        Capabilities:
            <<: *ApplicationCapabilities
```

在 Profiles 配置中，TwoOrgsOrdererGenesis、OrdererDefaults、SampleConsortium、TwoOrgsChannel 及 ApplicationDefaults 五个字段可以自定义成自己所搭建联盟链的相关名称，可以显得更加正式与严肃。但需要注意上下文对应，即前文改了字段名称，后文需要与之对应做出修改，以免出现无法找到相关配置的报错，且不容易排查。

其 Profiles-TwoOrgsOrdererGenesis-Orderer-Organizations-OrdererOrg 的名称最好自定义，官方给出的 configtx.yaml 中 OrdererOrg 只是样本名称，生产环境下最好使用自定义方式来命名。

其 Profiles-TwoOrgsOrdererGenesis-Consortiums-SampleConsortium-Organizations 下定义了 Peer 节点数量及 Peer 节点名称。而在 TwoOrgsChannel-Application-Organizations 的名称需要与之对应。

上述两种需自定义的节点服务名称均需要在 crypto-config.yaml 配置文件中进行体现且名称一致。

组织节点信息 Organizations 下定义的是所有的排序服务节点及普通的组织节点根服务器配置信息，此处配置同样需要与上述内容字段匹配，具体示例如下：

```
Organizations:

    # SampleOrg defines an MSP using the sampleconfig.  It should never be used
    # in production but may be used as a template for other definitions
    - &OrdererOrg
        # DefaultOrg defines the organization which is used in the sampleconfig
        # of the fabric.git development environment
        Name: OrdererOrg

        # ID to load the MSP definition as
        ID: OrdererMSP

        # MSPDir is the filesystem path which contains the MSP configuration
        MSPDir: crypto-config/ordererOrganizations/example.com/msp

    - &Org1
        # DefaultOrg defines the organization which is used in the sampleconfig
        # of the fabric.git development environment
        Name: Org1MSP

        # ID to load the MSP definition as
        ID: Org1MSP
```

```
        MSPDir: crypto-config/peerOrganizations/org1.example.com/msp

        AnchorPeers:
            # AnchorPeers defines the location of peers which can be used
            # for cross org gossip communication.  Note, this value is only
            # encoded in the genesis block in the Application section context
            - Host: peer0.org1.example.com
              Port: 7051

    - &Org2
        # DefaultOrg defines the organization which is used in the sampleconfig
        # of the fabric.git development environment
        Name: Org2MSP

        # ID to load the MSP definition as
        ID: Org2MSP

        MSPDir: crypto-config/peerOrganizations/org2.example.com/msp

        AnchorPeers:
            # AnchorPeers defines the location of peers which can be used
            # for cross org gossip communication.  Note, this value is only
            # encoded in the genesis block in the Application section context
            - Host: peer0.org2.example.com
              Port: 7051
```

在该配置中，Name 是给定的节点名称，ID 是即将加载的自定义 MSPID，MSPDir 则是定义包含 MSP 配置的文件系统路径。

Peer 节点中的 AnchorPeers 定义了节点的位置，可以用于跨组织的数据传播或同步，且这个值只在应用程序部分的上下文中被编码。

接下来是 Orderer 的配置，Orderer 指定了 Fabric 网络的启动类型、区块生成配置及排序服务的地址。这一块的配置相当重要，需要预估自己所搭建的联盟平台的使用量和具体的即时性需求。

Orderer 的具体配置示例如下所示：

```
Orderer: &OrdererDefaults

    # Orderer Type: The orderer implementation to start
    # Available types are "solo" and "kafka"
    OrdererType: kafka
```

```
    Addresses:
        - orderer.example.com:7050

    # Batch Timeout: The amount of time to wait before creating a batch
    BatchTimeout: 2s

    # Batch Size: Controls the number of messages batched into a block
    BatchSize:

        # Max Message Count: The maximum number of messages to permit in a batch
        MaxMessageCount: 10

        # Absolute Max Bytes: The absolute maximum number of bytes allowed for
        # the serialized messages in a batch.
        AbsoluteMaxBytes: 98 MB

        # Preferred Max Bytes: The preferred maximum number of bytes allowed for
        # the serialized messages in a batch. A message larger than the preferred
        # max bytes will result in a batch larger than preferred max bytes.
        PreferredMaxBytes: 512 KB

    # Organizations is the list of orgs which are defined as participants on
    # the orderer side of the network
    Organizations:
```

在 Orderer 中 OrdererType 定义了启动类型，从 Fabric 1.0 版本开始就默认了两种启动类型，分别是 Solo 和 Kafka，测试合约的情况下可以使用 Solo 类型，但实际生产上线时则必须使用 Kafka 来提高容错处理能力。

Addresses 是用来定义一组 Orderer 排序服务节点地址码，如果上述启动类型选择 Solo，则只需要一台 Orderer 排序服务节点即可。如果上述启动类型选择了 Kafka，则需要考虑具体生产所需的 Orderer 排序服务节点到底应该部署多少，理论上可以预先多设置额外的 Orderer 排序服务节点地址。这样，在排序服务节点服务器遇到硬件性能瓶颈的时候，可以直接动态新增，而不需要通过修改创世区块配置的方式手动新增。

在 Orderer 中的 MaxMessageCount、AbsoluteMaxBytes 和 BatchTimeout 算是一组组合参数，解释如下：

（1）BatchTimeout。批处理超时：在创建批处理之前等待的时间。

（2）MaxMessageCount。最大消息计数：批处理的最大消息数量

（3）AbsoluteMaxBytes。绝对最大字节：批中序列化消息的绝对最大字节数。

上述三点是生成一个区块的最低条件，以官方默认配置为例，如下所示：

```
BatchTimeout: 2s
    BatchSize:
        MaxMessageCount: 10
        AbsoluteMaxBytes: 98 MB
        PreferredMaxBytes: 512 KB
```

区块是在排序服务中生成，最终再通过广播的形式分发到各个 Peer 节点上实现数据同步，而区块什么时候生成以上述参数设置为主。如上述参数表示一个区块中只要达到了 10 条消息体，或者区块大小达到了 98MB，或者距离上一区块生成时间达到或超过 2s，满足三个中的任意条件即在排序服务节点中生成一个新的区块。

对于即时性要求较高的联盟可根据实际需求来设置 BatchTimeout 参数的值，而 MaxMessageCount 则建议不要采用默认设置，建议最低也设置为 100，否则最终同步的时候会造成同步区块数量较多。而同步区块中同样的头尾文件信息量要大于实际读写集，从而导致数据膨胀比较严重。区块中消息的绝对最大字节建议调小，按照程序员的习惯，可以设置为 32MB 或 64MB，一般情况下读写集的内容不会太大，32MB 足够使用。

在 Application 中的 Organizations 是组织的列表，它被定义为网络应用程序方面的参与者，按照官方案例中的配置不予配置即可，具体示例如下所示：

```
Application: &ApplicationDefaults

    # Organizations is the list of orgs which are defined as participants on
    # the application side of the network
    Organizations:
```

Capabilities 是 Fabric1.1 版本新推出来的配置，为了更加方便地认识它，这里将它的配置信息摘选出来，如下所示：

```
Capabilities:
    Global: &ChannelCapabilities
        V1_1: true
    Orderer: &OrdererCapabilities
        V1_1: true
    Application: &ApplicationCapabilities
        V1_1: true
```

其中，Global: &ChannelCapabilities 按照其字面意思是全局频道功能配置，频道功能必须同时适用并支持排序服务节点以及 Peer 节点。并以达到前述目的而将该功能的值设置为 true（对于全局频道配置来说，v1_1 是一种针对所有运行 v1.0.x 的排序服务节点和 Peer 节点所期望的行为标记，修改后将导致不兼容。应该把这个标志设置为 true）。

Orderer: &OrdererCapabilities 按照其字面意思是排序服务功能配置，Orderer 功能只适用于

排序服务节点，并且可以安全地操作，而不需要考虑升级 Peer 节点。需要将该能力的值设置为 true（对于运行 v1.0.x 的所有排序服务节点来说，Order 是一个标记为所有行为的标记，修改后会导致不兼容。应该把这个标志设置为 true）。

Application: &ApplicationCapabilities 按照其字面意思是应用功能配置，应用程序功能只适用于 Peer 节点网络，并且可以安全地操作，而不需要考虑升级排序服务节点。需要将该能力的值设置为 true（对于运行 v1.0.x 的所有 Peer 节点来说，应用程序是一个捕获的所有行为的标记，修改后会导致不兼容。应该把这个标志设置为 true）。

现在继续回到 generateArtifacts.sh 脚本中的 generateChannelArtifacts 方法，在了解了上述配置文件及参数含义后，对 generateChannelArtifacts 方法中的各项命令应该会有更为深刻的认识。该方法执行完成后，将依次生成创世区块、频道配置文件、组织 1Peer 节点文件及组织 2Peer 节点文件。

其中，创世区块文件需要在启动排序服务节点的时候使用，频道配置文件需要在 Peer 节点执行频道创建及加入等操作的时候使用，而不同组织的 Peer 节点文件会需要在更新全局配置文件的时候使用。

当 generateChannelArtifacts 执行完成后，generateArtifacts.sh 脚本中所有的可执行方法都已经完毕，此时再回头查阅调用该脚本的源文件 network_setup.sh 脚本。

在 network_setup.sh 脚本文件中的 networkUp 方法执行完 generateArtifacts.sh 后会继续判断即将启动的 Compose 文件是否是 CouchDB 的 Compose，而在启动 network_setup.sh 脚本时并未指定 CouchDB，因此根据脚本含义将启动 docker-compose-cli.yaml 文件。

到此为止，network_setup.sh 脚本的工作就结束了，接下是对即将启动 docker-compose-cli.yaml 进行进一步的解析。

### 3.3.2 容器启动配置文件

如下所示是 docker-compose-cli.yaml 文件：

```
# Copyright IBM Corp. All Rights Reserved.
#
# SPDX-License-Identifier: Apache-2.0
#

version: '2'

services:

  orderer.example.com:
    extends:
```

```
        file:  base/docker-compose-base.yaml
        service: orderer.example.com
      container_name: orderer.example.com

    peer0.org1.example.com:
      container_name: peer0.org1.example.com
      extends:
        file:  base/docker-compose-base.yaml
        service: peer0.org1.example.com

    peer1.org1.example.com:
      container_name: peer1.org1.example.com
      extends:
        file:  base/docker-compose-base.yaml
        service: peer1.org1.example.com

    peer0.org2.example.com:
      container_name: peer0.org2.example.com
      extends:
        file:  base/docker-compose-base.yaml
        service: peer0.org2.example.com

    peer1.org2.example.com:
      container_name: peer1.org2.example.com
      extends:
        file:  base/docker-compose-base.yaml
        service: peer1.org2.example.com

    cli:
      container_name: cli
      image: hyperledger/fabric-tools
      tty: true
      environment:
        - GOPATH=/opt/gopath
        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_PEER_ID=cli
        - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
        - CORE_PEER_LOCALMSPID=Org1MSP
        - CORE_PEER_TLS_ENABLED=true
        - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
```

```
        - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric
/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.cr
t
        - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      command: /bin/bash -c './scripts/script.sh ${CHANNEL_NAME}; sleep $TIMEOUT'
      volumes:
          - /var/run/:/host/var/run/
          - ../chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/examples/
chaincode/go
          - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
          - ./scripts:/opt/gopath/src/github.com/hyperledger/fabric/peer/scripts/
          - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/
channel-artifacts
      depends_on:
        - orderer.example.com
        - peer0.org1.example.com
        - peer1.org1.example.com
        - peer0.org2.example.com
        - peer1.org2.example.com
```

从配置文件中不难看出，排序服务节点继承了 base/docker-compose-base.yaml 中的 orderer.example.com 属性，而其他四个 Peer 节点继承了 base/docker-compose-base.yaml 中的与之容器名称对应的属性。

在上述文件所继承的文件属性中，关键属性如下：

（1）environment 是当前所配置的外界的环境变量。

（2）working_dir 是当前容器启动后的工作路径。

（3）volumes 是外界物理机路径挂载或指引到容器内的路径。

（4）ports 是指定当前容器启动后映射到物理机上的端口号。

（5）depends_on 是指定当前自身容器启动所依赖的启动容器对象。

在 docker-compose-cli.yaml 文件中，官方的节点配置信息基本已经满足实际生产应用，而真正使用这些服务节点进行数据维护和管理的则是交由客户端或 SDK 来执行。这里也就是配置文件中的 cli。

关于排序服务节点、Peer 节点以及 cli 中的属性值的具体意义将在后续的章节中讲述，这里暂时不做赘述。直接看 cli 中 command 属性，在 command 中指定当 cli 容器启动后会执行当前目录中 scripts 目录下的 script.sh 脚本。

### 3.3.3　Fabric网络解析

script.sh 脚本的执行则是 e2e_cli 中真正地对 Peer 节点、频道以及合约的集合操作演示，官方出版的该脚本试图通过这样的方式来指导开发人员尽快入门。因此，script.sh 脚本中的每一步操作对刚入门的开发人员来说都极具参考价值。

首先来看该脚本内方法执行顺序，如下所示：

```
## Create channel
echo "Creating channel..."
createChannel

## Join all the peers to the channel
echo "Having all peers join the channel..."
joinChannel

## Set the anchor peers for each org in the channel
echo "Updating anchor peers for org1..."
updateAnchorPeers 0
echo "Updating anchor peers for org2..."
updateAnchorPeers 2

## Install chaincode on Peer0/Org1 and Peer2/Org2
echo "Installing chaincode on org1/peer0..."
installChaincode 0
echo "Install chaincode on org2/peer2..."
installChaincode 2

#Instantiate chaincode on Peer2/Org2
echo "Instantiating chaincode on org2/peer2..."
instantiateChaincode 2

#Query on chaincode on Peer0/Org1
echo "Querying chaincode on org1/peer0..."
chaincodeQuery 0 100

#Invoke on chaincode on Peer0/Org1
echo "Sending invoke transaction on org1/peer0..."
chaincodeInvoke 0

## Install chaincode on Peer3/Org2
echo "Installing chaincode on org2/peer3..."
installChaincode 3
```

```
#Query on chaincode on Peer3/Org2, check if the result is 90
echo "Querying chaincode on org2/peer3..."
chaincodeQuery 3 90
```

该脚本设计 Peer 节点及排序服务节点的操作分别有如下九个步骤：

（1）createChannel。根据之前在 generateArtifacts.sh 脚本中通过 configtx.yaml 配置文件生成的频道文件创建频道。

（2）joinChannel。Peer 节点加入指定频道。

（3）updateAnchorPeers。为频道中的每个组织设置 Peer 节点。

（4）installChaincode。在 Peer0/Org1 和 Peer2/Org2 上安装智能合约。

（5）instantiateChaincode。在 Peer2/Org2 上对智能合约进行实例化操作。

（6）chaincodeQuery。在 Peer0/Org1 上执行智能合约中的查询方法。

（6）chaincodeInvoke。在 Peer0/Org1 上执行智能合约中的交易方法。

（8）installChaincode。在 Peer3/Org2 上安装智能合约。

（9）chaincodeQuery。在 Peer3/Org2 上执行智能合约中的查询方法。

按照上面讲述的方法执行顺序，首先执行的 createChannel 方法具体如下所示：

```
createChannel() {
 setGlobals 0

        if [ -z "$CORE_PEER_TLS_ENABLED" -o "$CORE_PEER_TLS_ENABLED" = "false" ];
then
        peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel
-artifacts/channel.tx >&log.txt
    else
        peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channe
l-artifacts/channel.tx --tls $CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA >&log.txt
    fi
    res=$?
    cat log.txt
    verifyResult $res "Channel creation failed"
    echo "===================== Channel \"$CHANNEL_NAME\" is created successfully
===================== "
    echo
}
```

其中 setGlobals 是一个设置全局环境参数的方法，该方法能够指定当前可执行 Peer 节点对象中的 CORE_PEER_LOCALMSPID、CORE_PEER_TLS_ROOTCERT_FILE 及 CORE_PEER_MSPCONFIGPATH 三个环境参数，同时指定 CORE_PEER_ADDRESS 访问地址。即通过 setGlobals 方法可以在客户端中指定当前所操作节点的具体对象。

在 createChannel 方法中指定了 peer0org0 节点来执行频道创建操作，具体命令格式如下所示：

```
    peer channel create -o orderer.example.com:7050（指定排序服务地址） -c $CHANNEL_NAME
（频道名称） -f ./channel-artifacts/channel.tx（当前频道创建之前所绑定的频道文件）
```

频道创建成功后会向下继续执行 joinChannel 方法，该方法具体如下所示：

```
    joinChannel () {
     for ch in 0 1 2 3; do
          setGlobals $ch
          joinWithRetry $ch
          echo "==================== PEER$ch joined on the channel \"$CHANNEL_NAME\"
==================== "
          sleep 2
          echo
     done
    }
```

在该方法中有一个循环，通过设置不同的当前 Peer 节点来调用 joinWithRetry 方法执行频道加入的操作，joinWithRetry 方法示例如下：

```
    joinWithRetry () {
     peer channel join -b $CHANNEL_NAME.block  >&log.txt
     res=$?
     cat log.txt
     if [ $res -ne 0 -a $COUNTER -lt $MAX_RETRY ]; then
          COUNTER=` expr $COUNTER + 1`
          echo "PEER$1 failed to join the channel, Retry after 2 seconds"
          sleep 2
          joinWithRetry $1
     else
          COUNTER=1
     fi
           verifyResult $res "After $MAX_RETRY attempts, PEER$ch has failed to Join the
Channel"
    }
```

该方法中的核心命令如下：

```
    peer channel join -b $CHANNEL_NAME.block
```

在执行加入频道成功后将继续循环，如果失败则会重新尝试加入频道的操作，持续失败 5 次则终止脚本并抛出错误提示。

当 4 个节点加入频道的操作全部完成后，会继续向下执行两次 updateAnchorPeers 方法，以便更新各个组织锚节点的操作。锚节点定义了 Peer 节点的位置，用于跨组织间的节点进行

Gossip 通信，实现同步。

相对于 Org1，peer0.org1 是锚节点，需要切换到 peer0.org1 服务器上并更新锚节点；同样，相对于 Org2，peer0.org2 是锚节点，需要切换到 peer0.org2 服务器上并更新锚节点。该方法具体示例如下：

```
   updateAnchorPeers() {
         PEER=$1
         setGlobals $PEER

         if [ -z "$CORE_PEER_TLS_ENABLED" -o "$CORE_PEER_TLS_ENABLED" = "false" ];
then
        peer channel update -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel
-artifacts/${CORE_PEER_LOCALMSPID}anchors.tx >&log.txt
    else
        peer channel update -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel
-artifacts/${CORE_PEER_LOCALMSPID}anchors.tx --tls $CORE_PEER_TLS_ENABLED --cafile
$ORDERER_CA >&log.txt
    fi
    res=$?
    cat log.txt
    verifyResult $res "Anchor peer update failed"
    echo "===================== Anchor peers for org \"$CORE_PEER_LOCALMSPID\" on
\"$CHANNEL_NAME\" is updated successfully ===================== "
    sleep 5
    echo
   }
```

锚节点更新完毕后，将会执行 installChaincode 方法，在 Peer0/Org1 和 Peer2/Org2 上安装智能合约，该方法与更新锚节点方法一样，会被调用两次，以便实现在两个锚节点上都安装指定智能合约。具体实现如下：

```
   installChaincode () {
    PEER=$1
    setGlobals $PEER
    peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/
chaincode/go/chaincode_example02 >&log.txt
    res=$?
    cat log.txt
         verifyResult $res "Chaincode installation on remote peer PEER$PEER has
Failed"
    echo "===================== Chaincode is installed on remote peer PEER$PEER
===================== "
    echo
   }
```

其中，核心安装智能合约方法命令如下所示：

```
peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/
chaincode/go/chaincode_example02
```

智能合约需要在每一台 Peer 节点服务器上执行一次安装操作，而智能合约的实例化操作与安装合约不同，实例化只需要执行一次即可。即假定有 4 台 Peer 节点服务器，如果这 4 台 Peer 节点服务器都有运行同一智能合约的需求，则都需要执行一次智能合约的安装操作，但实例化智能合约只需要在其中任意一台就能完成，多次实例化并没有额外的意义。

---

**注意：** 升级智能合约与实例化智能合约一样，都只需要在任意一台服务器进行操作。

---

智能合约安装完成后继续下一步就是实例化该智能合约，具体执行 instantiateChaincode 方法来进行实例化操作，方法示例如下：

```
instantiateChaincode () {
 PEER=$1
 setGlobals $PEER
 # while 'peer chaincode' command can get the orderer endpoint from the peer (if
join was successful),
 # lets supply it directly as we know it using the "-o" option
 if [ -z "$CORE_PEER_TLS_ENABLED" -o "$CORE_PEER_TLS_ENABLED" = "false" ]; then
     peer chaincode instantiate -o orderer.example.com:7050 -C $CHANNEL_NAME -n
mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "OR
 ('Org1MSP.member','Org2MSP.member')" >&log.txt
 else
     peer chaincode instantiate -o orderer.example.com:7050 --tls $CORE_PEER_TLS_
ENABLED --cafile $ORDERER_CA -C $CHANNEL_NAME -n mycc -v 1.0 -c '{"
Args":["init","a","100","b","200"]}' -P "OR
 ('Org1MSP.member','Org2MSP.member')" >&log.txt
 fi
 res=$?
 cat log.txt
 verifyResult $res "Chaincode instantiation on PEER$PEER on channel '$CHANNEL_NAME'
failed"
 echo "===================== Chaincode Instantiation on PEER$PEER on channel
'$CHANNEL_NAME' is successful ===================== "
 echo
}
```

该方法是在 Peer2/Org2 上对智能合约进行实例化操作，且该方法在 script.sh 脚本中仅执行了一次。

同时，在执行智能合约实例化操作的时候需要确认背书方案。所谓背书，即对当前合约执行写入的权限控制方案，背书支持 and 和 or 的单例或混合书写方案，是在宏观上确保联盟链安全性的最大外围设置。有关背书的详细内容，后续章节中会进行介绍。

在该方法中执行智能合约实例化操作的关键命令如下所示：

```
   peer chaincode instantiate -o orderer.example.com:7050 -C $CHANNEL_NAME -n mycc -v
1.0 -c '{"Args":["init","a","100","b","200"]}' -P "OR
    ('Org1MSP.member','Org2MSP.member')"
```

其中，init 是执行智能合约实例化或升级时候初始化方法，根据需求选择性使用，也可以为空。

最后的 OR ('Org1MSP.member','Org2MSP.member')即表示背书操作，这里简单表述下该背书的含义，即 Org1 和 Org2 两个组织中成员任意一方执行区块写入操作都判定合法写入。

前面 4 步是比较容易出错的步骤，即便是后续通过手动配置实现也容易出错。当创建频道、加入频道、安装智能合约及实例化智能合约全部完成后，后续操作将会直接与智能合约挂钩，基本很少再出现环境配置等方面不容易排查的错误。

实例化智能合约之后，script.sh 脚本的下一步就开始执行 chaincodeQuery 方法对智能合约进行一次查询操作。具体方法示例如下：

```
   chaincodeQuery () {
     PEER=$1
     echo "===================== Querying on PEER$PEER on channel '$CHANNEL_NAME'...
===================== "
     setGlobals $PEER
     local rc=1
     local starttime=$(date +%s)

     # continue to poll
     # we either get a successful response, or reach TIMEOUT
     while test "$(($(date +%s)-starttime))" -lt "$TIMEOUT" -a $rc -ne 0
     do
        sleep 3
        echo "Attempting to Query PEER$PEER ...$(($(date +%s)-starttime)) secs"
        peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query",
"a"]}' >&log.txt
        test $? -eq 0 && VALUE=$(cat log.txt | awk '/Query Result/ {print $NF}')
        test "$VALUE" = "$2" && let rc=0
     done
     echo
     cat log.txt
     if test $rc -eq 0 ; then
```

```
    echo "===================== Query on PEER$PEER on channel '$CHANNEL_NAME' is
successful ===================== "
     else
    echo "!!!!!!!!!!!!!!! Query result on PEER$PEER is INVALID !!!!!!!!!!!!!!!!"
          echo "================== ERROR !!! FAILED to execute End-2-End Scenario
=================="
    echo
    exit 1
     fi
   }
```

在该方法中执行智能合约查询操作的关键命令如下所示：

```
   peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}
```

查询方法其实与 Invoke 写入方法可以一起解读，在 chaincodeInvoke 方法中执行智能合约写入操作的关键命令如下：

```
   peer chaincode invoke -o orderer.example.com:7050 -C $CHANNEL_NAME -n mycc -c
'{"Args":["invoke","a","b","10"]}'
```

在上述两条命令中，分别执行了 query 和 invoke 指令，query 是执行合约但不发送读写集到排序服务，invoke 是既执行合约又同时把 Peer 节点返回的读写集发送至排序服务形成数据写入操作，最终生成区块并同步到所有节点中。

对于智能合约操作的这一块内容，后续的章节会有所介绍。另外，关于该合约所执行的具体内容需要阅读合约类文件，具体在 installChaincode 方法中已经给出了该合约的具体路径，但需要与 docker-compose-cli.yaml 中 cli 的映射路径做对比发现物理机的真实路径。即当前目录上一级 examples 目录下/chaincode/go/chaincode_example02 目录下，可以通过阅读该合约预先了解合约的含义。

chaincodeInvoke 方法的具体示例如下：

```
   chaincodeInvoke () {
    PEER=$1
    setGlobals $PEER
    # while 'peer chaincode' command can get the orderer endpoint from the peer (if
join was successful),
    # lets supply it directly as we know it using the "-o" option
    if [ -z "$CORE_PEER_TLS_ENABLED" -o "$CORE_PEER_TLS_ENABLED" = "false" ]; then
        peer chaincode invoke -o orderer.example.com:7050 -C $CHANNEL_NAME -n mycc -c
'{"Args":["invoke","a","b","10"]}' >&log.txt
    else
        peer chaincode invoke -o orderer.example.com:7050  --tls
$CORE_PEER_TLS_ENABLED --cafile $ORDERER_CA -C $CHANNEL_NAME -n mycc -c
```

```
'{"Args":["invoke","a","b","10"]}' >&log.txt
    fi
    res=$?
    cat log.txt
    verifyResult $res "Invoke execution on PEER$PEER failed "
    echo "===================== Invoke transaction on PEER$PEER on channel
'$CHANNEL_NAME' is successful ===================== "
    echo
   }
```

## 3.4 本章小结

通过本章概述，对于 Fabric 网络启动的先决条件有了基本的了解；通过执行一次 e2e_cli 案例，并针对其中容易出现的错误进行了简单分析；最后，综合 e2e_cli 案例的内容相当于变相带着手动参与了一次简化版的 Fabric 网络过程。

很多 HyperLedger Fabric 开发人员在入门的时候都会尝试 e2e_cli 案例，至少也会以这个案例来检查当前服务器的 Fabric 网络环境是否合适。但还是有相当一部分人会卡在该案例上。

其实，如果没有按照第 1 章进行环境整理操作的话，可能还会出现 Linux 内核 Bug，该 Fabric 网络集群测试环境在 Linux 内核低版本上可能会出现问题，这是旧版内核的 Bug，比如在执行 bash network_setup.sh up 的时候会出现如图 3-12 所示错误。

这是一个很隐晦的错误，如果对 Linux 了解不够或是网络检索方案不当，会给初学者带来很大困扰。

但案例始终是案例，学习 HyperLedger Fabric 不仅仅是跑通一个案例，如果案例无法跑通，可以通过阅读执行该案例的脚本来深入学习，且通过阅读脚本学习效果会比跑通一个案例要好得多。

e2e_cli 案例几乎涉及了 Fabric 网络中超过 80%的情况，有种案例即生产的即视感，其中所涉及的节点证书文件生成、节点用户管理和节点频道管理，后续还有频道及智能合约相关操作，该脚本可以说是入门学习的必备。

```
===================== Chaincode is installed on remote peer PEER2 =====================

Instantiating chaincode on org2/peer2...
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2
CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.exam
CORE_PEER_LOCALMSPID=Org2MSP
CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.exa
CORE_PEER_TLS_ENABLED=true
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.exa
CORE_PEER_ID=cli
CORE_LOGGING_LEVEL=DEBUG
CORE_PEER_ADDRESS=peer0.org2.example.com:7051
2017-08-10 06:14:50.896 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2017-08-10 06:14:50.896 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2017-08-10 06:14:50.899 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2017-08-10 06:14:50.899 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2017-08-10 06:14:50.900 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A95070A6708031A0C08DAF3AFCC051
2017-08-10 06:14:50.900 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 2AC9638E8FE710404A5F6B61990224CEF2
Error: Error endorsing chaincode: rpc error: code = Unknown desc = Error starting container: API error (500)
nux.go:339: container init caused \\\"read init-p: connection reset by peer\\\"\"\n"}

Usage:
  peer chaincode instantiate [flags]

Flags:
  -C, --channelID string   The channel on which this command should be executed (default "testchainid")
  -c, --ctor string        Constructor message for the chaincode in JSON format (default "{}")
  -E, --escc string        The name of the endorsement system chaincode to be used for this chaincode
  -l, --lang string        Language the chaincode is written in (default "golang")
  -n, --name string        Name of the chaincode
  -P, --policy string      The endorsement policy associated to this chaincode
  -v, --version string     Version of the chaincode specified in install/instantiate/upgrade commands
  -V, --vscc string        The name of the verification system chaincode to be used for this chaincode

Global Flags:
      --cafile string              Path to file containing PEM-encoded trusted certificate(s) for the orderi
      --logging-level string       Default logging level and overrides, see core.yaml for full syntax
  -o, --orderer string             Ordering service endpoint
      --test.coverprofile string   Done (default "coverage.cov")
      --tls                        Use TLS when communicating with the orderer endpoint

!!!!!!!!!!!!!!! Chaincode instantiation on PEER2 on channel 'mychannel' failed !!!!!!!!!!!!!!!!
==================== ERROR !!! FAILED to execute End-2-End Scenario ====================
```

图3-12 内核Bug

# 第 4 章　部署单机多节点网络

本章单机多节点网络的部署是应对第 3 章中的 e2e_cli 网络案例，通过纯手动操作的方式运行一遍 Fabric 网络，加深对 Fabric 网络部署的认识，同时也为后续集群网络的部署打下基础。

如果想顺利地进行本章节阅读，需要先掌握第 1 章和第 2 章的内容。也就是说 Fabric 最基础的网络环境部署要能够符合条件，如果能把第 3 章中讲述的 e2e_cli 案例跑通，那无疑是最好，能够更为快速地开始本章学习。

在本次单机多节点网络部署时会采用与 e2e_cli 案例类似的方案，即一个排序服务节点与两个组织下的 Peer 节点进行组网，其中每个组织都包含两个 Peer 节点，简要网络拓扑如图 4-1 所示。

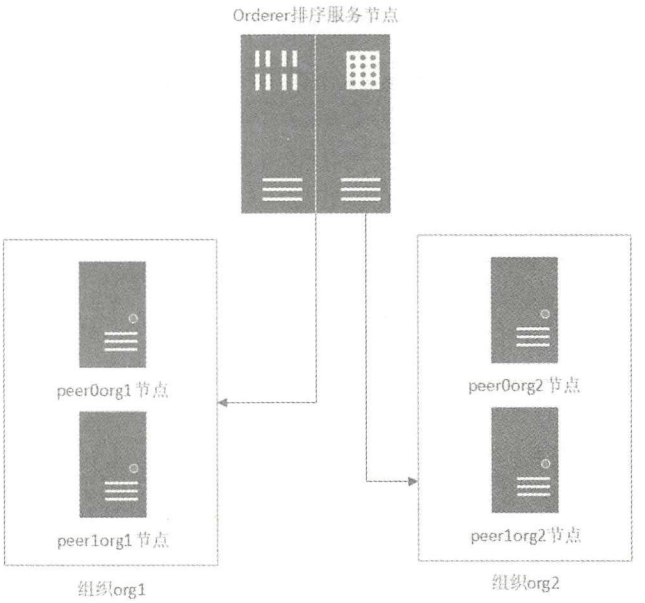


图4–1　简要网络拓扑

本次部署的启动顺序是先启动一个 Orderer 排序服务节点，随后启动 peer0.org1 节点并在该节点执行频道及合约相关操作，测试通过后再启动 peer0.org2 节点，并使得新启动的节点加入到网络及网络所属频道中，实现一个伪动态加入。

## 4.1　生成证书文件

通过第 1 章与第 2 章中的步骤操作，可以完成环境整理、环境部署、镜像下载与源码处理，最终可以得到一个完全干净的空服务器状态。如果是运行过第 3 章所写的 e2e_cli 案例，则执行如下命令即可做到环境清理：

```
bash network_setup.sh down
```

最终得到如图 4-2 所示结果。

```
[root@VM_139_63_centos e2e_cli]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
[root@VM_139_63_centos e2e_cli]# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
[root@VM_139_63_centos e2e_cli]# docker images
REPOSITORY                     TAG                 IMAGE ID            CREATED             SIZE
hyperledger/fabric-zookeeper   latest              1ce465be7112        13 days ago         1.39GB
hyperledger/fabric-zookeeper   x86_64-0.4.7        1ce465be7112        13 days ago         1.39GB
hyperledger/fabric-kafka       latest              4fed436fc0a0        13 days ago         1.4GB
hyperledger/fabric-kafka       x86_64-0.4.7        4fed436fc0a0        13 days ago         1.4GB
hyperledger/fabric-couchdb     latest              35228d48a25a        13 days ago         1.56GB
hyperledger/fabric-couchdb     x86_64-0.4.7        35228d48a25a        13 days ago         1.56GB
hyperledger/fabric-baseos      x86_64-0.4.7        c0e784934c4e        13 days ago         152MB
hyperledger/fabric-ca          latest              72617b4fa9b4        3 weeks ago         299MB
hyperledger/fabric-ca          x86_64-1.1.0        72617b4fa9b4        3 weeks ago         299MB
hyperledger/fabric-tools       latest              b7bfddf508bc        3 weeks ago         1.46GB
hyperledger/fabric-tools       x86_64-1.1.0        b7bfddf508bc        3 weeks ago         1.46GB
hyperledger/fabric-orderer     latest              ce0c810df36a        3 weeks ago         180MB
hyperledger/fabric-orderer     x86_64-1.1.0        ce0c810df36a        3 weeks ago         180MB
hyperledger/fabric-peer        latest              b023f9be0771        3 weeks ago         187MB
hyperledger/fabric-peer        x86_64-1.1.0        b023f9be0771        3 weeks ago         187MB
hyperledger/fabric-ccenv       x86_64-1.1.0        c8b4909d8d46        3 weeks ago         1.39GB
hyperledger/fabric-baseos      x86_64-0.4.6        220e5cf3fb7f        7 weeks ago         151MB
[root@VM_139_63_centos e2e_cli]# go version
go version go1.10 linux/amd64
[root@VM_139_63_centos e2e_cli]# docker-compose version
docker-compose version 1.19.0, build 9e633ef
docker-py version: 2.7.0
CPython version: 2.7.13
OpenSSL version: OpenSSL 1.0.1t  3 May 2016
```

图4-2　环境信息

在已经加载好的 HyperLedger Fabric1.1 源码下创建一个名为 aberic 的目录，该目录为当前即将运行的单机多节点项目目录，FTP 完整目录如图 4-3 所示。

| 名称 | 大小 | 类型 | 修改时间 | 属性 | 所有者 |
|---|---|---|---|---|---|
| .. | | | | | |
| aberic | | 文件夹 | 2018/3/22, 12:58 | drwxrw... | root |
| bccsp | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| bddtests | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| common | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| core | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| devenv | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| docs | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| events | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| examples | | 文件夹 | 2018/4/8, 13:55 | drwxr-... | root |
| gossip | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| gotools | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| idemix | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| images | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| msp | | 文件夹 | 2018/3/20, 16:26 | drwxr-... | root |
| orderer | | 文件夹 | 2018/3/20, 16:27 | drwxr-... | root |
| peer | | 文件夹 | 2018/3/20, 16:27 | drwxr-... | root |
| proposals | | 文件夹 | 2018/3/20, 16:27 | drwxr-... | root |
| protos | | 文件夹 | 2018/3/20, 16:27 | drwxr-... | root |
| release | | 文件夹 | 2018/4/6, 21:19 | drwxr-... | root |
| release_notes | | 文件夹 | 2018/3/20, 16:27 | drwxr-... | root |
| sampleconfig | | 文件夹 | 2018/3/20, 16:27 | drwxr-... | root |
| scripts | | 文件夹 | 2018/3/20, 16:27 | drwxr-... | root |
| test | | 文件夹 | 2018/3/20, 16:27 | drwxr-... | root |
| unit-test | | 文件夹 | 2018/3/20, 16:27 | drwxr-... | root |
| vendor | | 文件夹 | 2018/3/20, 16:28 | drwxr-... | root |
| CHANGELOG.md | 464KB | MD 文件 | 2018/3/20, 16:26 | -rw-r--r-- | root |
| ci.properties | 13 Bytes | PROPE... | 2018/3/20, 16:26 | -rw-r--r-- | root |

图4-3　完整目录信息

在 aberic 目录中需要将第 3 章提到的 Fabric1.1 版本的平台特定二进制文件上传至该目录下，这些文件通过下载后会自动解压在一个名为 bin 的目录下。

而本章所使用的 configtx.yaml 及 crypto-config.yaml 配置文件皆来自 Fabric1.0 源码中的版本，即非 Kafka 启动类型，与第 3 章采用的 e2e_cli 方案相同。可以按照第 3 章所述方案提前准备好这两份文件并上传至 aberic 项目目录下，上传完成后 FTP 目录如图 4-4 所示。

/home/docker/github.com/hyperledger/fabric/aberic

| 名称 | 大小 | 类型 | 修改时间 | 属性 | 所有者 |
|---|---|---|---|---|---|
| .. | | | | | |
| bin | | 文件夹 | 2018/3/20, 17:48 | drwxr-... | root |
| configtx.yaml | 5KB | YAML ... | 2018/3/20, 16:37 | -rw-r--r-- | root |
| crypto-config.yaml | 4KB | YAML ... | 2018/3/20, 16:37 | -rw-r--r-- | root |

图4-4　aberic目录

接下来可以开始生成所需证书文件了，执行相关命令需要指定执行文件的路径，为了方

便，直接进入 aberic 项目目录下进行操作，随后执行如下命令生成项目所需文件：

```
./bin/cryptogen generate --config=./crypto-config.yaml
```

整体操作结果如图 4-5 所示。

```
[root@VM_139_63_centos ~]# cd /home/docker/github.com/hyperledger/fabric/aberic/
[root@VM_139_63_centos aberic]# ./bin/cryptogen generate --config=./crypto-config.yaml
org1.example.com
org2.example.com
```

图4-5　生成节点文件

在此过程中可能会提示权限不足，如图 4-6 所示。

```
[root@p0-chaincode-app~10.130.116.8 /opt/gopath/src/tk/bin]# ./cryptogen generate --config=../crypto-config.yaml
-bash: ./cryptogen: Permission denied
```

图4-6　权限不足提示

这里使用 chmod +x 命令赋权即可，具体可参考第 3.2 节中的解决方案。

完成之后会在 bin 目录下生成一个新的目录 crypto-config，其中会有 ordererOrganizations 和 peerOrganizations 两个目录，具体结果如图 4-7 所示。

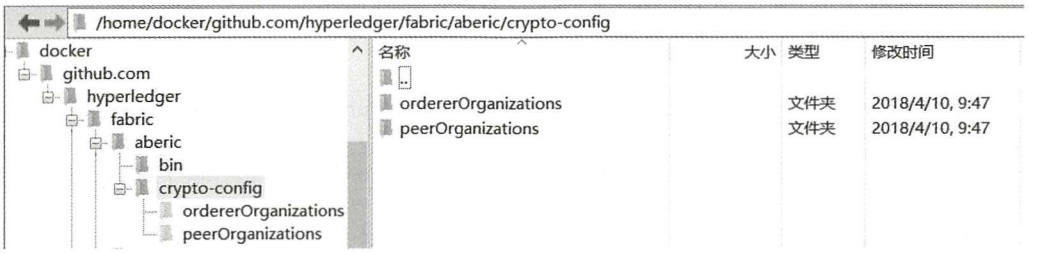


图4-7　crypto-config目录

接下来，使用 configtxgen 工具执行 configtx.yaml 文件以创建 orderer Genesis block，在此之前需要为 configtxgen 工具指定 configtx.yaml 文件的路径，设置环境变量，执行如下命令：

```
export FABRIC_CFG_PATH=$PWD
```

随后可以打印查看该目录是否正确，具体执行命令如下：

```
echo $PWD
```

结果如图 4-8 所示。

```
[root@VM_139_63_centos aberic]# export FABRIC_CFG_PATH=$PWD
[root@VM_139_63_centos aberic]# echo $PWD
/home/docker/github.com/hyperledger/fabric/aberic
```

图4-8　环境设置

接下来，根据 configtx.yaml 生成创世区块及频道认证文件，具体命令及执行后结果可能出现如下所示：

```
    [root@VM_139_63_centos aberic]# ./bin/configtxgen -profile TwoOrgsOrdererGenesis -
outputBlock ./channel-artifacts/genesis.block
    2018-04-10 10:13:26.239 CST [common/tools/configtxgen] main -> INFO 001 Loading
configuration
    2018-04-10 10:13:26.252 CST [common/tools/configtxgen] doOutputBlock -> INFO 002
Generating genesis block
    2018-04-10 10:13:26.252 CST [common/tools/configtxgen] doOutputBlock -> INFO 003
Writing genesis block
    2018-04-10 10:13:26.253 CST [common/tools/configtxgen] main -> CRIT 004 Error on
outputBlock: Error writing genesis block: open ./channel-artifacts/genesis.block: no
such file or directory
```

这里出现了一个小问题，提示没有该文件或文件夹。channel-artifacts 是一个文件夹。在 /home/docker/github.com/hyperledger/fabric/aberic 目录下手动创建一个 channel-artifacts 文件夹，随后再次运行上述命令，具体命令及结果如下所示：

```
    [root@VM_139_63_centos aberic]# ./bin/configtxgen -profile TwoOrgsOrdererGenesis -
outputBlock ./channel-artifacts/genesis.block
    2018-04-10 10:38:15.297 CST [common/tools/configtxgen] main -> INFO 001 Loading
configuration
    2018-04-10 10:38:15.309 CST [common/tools/configtxgen] doOutputBlock -> INFO 002
Generating genesis block
    2018-04-10 10:38:15.309 CST [common/tools/configtxgen] doOutputBlock -> INFO 003
Writing genesis block
```

通过 FTP 可以观察到该目录下已经创建出创世区块，如图 4-9 所示。

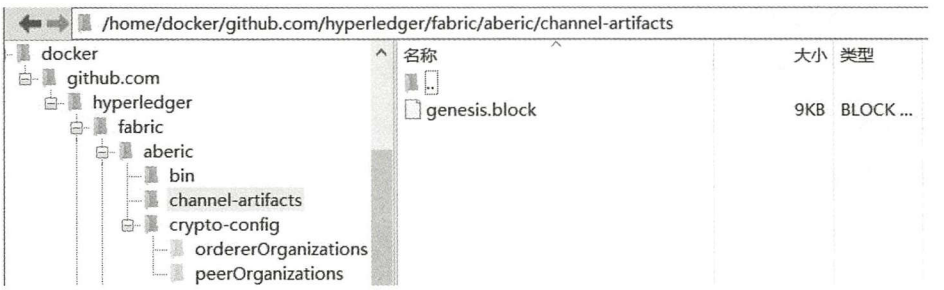


图4-9　创世区块

创世区块是为了 Orderer 排序服务启动时用到的，Peer 节点在启动后需要创建 Channel 的配置文件在这里也一并生成，执行具体命令和综合结果如下：

```
   [root@VM_139_63_centos aberic]# ./bin/configtxgen -profile TwoOrgsChannel -
outputCreateChannelTx ./channel-artifacts/mychannel.tx -channelID mychannel
   2018-04-10 10:43:29.935 CST [common/tools/configtxgen] main -> INFO 001 Loading
configuration
   2018-04-10 10:43:29.946 CST [common/tools/configtxgen] doOutputChannelCreateTx ->
INFO 002 Generating new channel configtx
   2018-04-10 10:43:29.974 CST [common/tools/configtxgen] doOutputChannelCreateTx ->
INFO 003 Writing new channel tx
```

再次通过 FTP 观察，可以发现在当前目录 channel-artifacts 目录下新建了一个名为 mychannel.tx 的频道文件。

该命令是生成了一个 channelID 为 mychannel 的 tx 文件（文件名称根据实际需求来取），通过该文件，peer 可以执行 channel 的创建工作，后面会提到。

## 4.2　部署Orderer节点

单机多节点部署，至少本章采用的启动类型是 Solo，而非 Kafka。如果开始考虑 Kafka 做集群，相信本章的内容已经不再适合你了，本章偏基础，可以阅读后面的 Kafka 集群部署章节。

首先需要编写一份 docker-orderer.yaml 启动文件，源码如下：

```
version: '2'

services:

  orderer.example.com:
    container_name: orderer.example.com
    image: hyperledger/fabric-orderer
    environment:
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
      # - ORDERER_GENERAL_LOGLEVEL=error
      - ORDERER_GENERAL_LOGLEVEL=debug
      - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
      - ORDERER_GENERAL_LISTENPORT=7050
      #- ORDERER_GENERAL_GENESISPROFILE=AntiMothOrdererGenesis
      - ORDERER_GENERAL_GENESISMETHOD=file
      - ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block
      - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
```

```
        - ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
        #- ORDERER_GENERAL_LEDGERTYPE=ram
        #- ORDERER_GENERAL_LEDGERTYPE=file
        # enabled TLS
        - ORDERER_GENERAL_TLS_ENABLED=false
        - ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
        - ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
        - ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric
      command: orderer
      volumes:
      - ./channel-
artifacts/genesis.block:/var/hyperledger/orderer/orderer.genesis.block
      - ./crypto-config/ordererOrganizations/example.com/orderers/orderer.example.
com/msp:/var/hyperledger/orderer/msp
      - ./crypto-config/ordererOrganizations/example.com/orderers/orderer.example
在.com/tls/:/var/hyperledger/orderer/tls
      networks:
        default:
          aliases:
            - aberic
      ports:
        - 7050:7050
```

这份启动文件中需要注意的是 Orderer 排序服务监听的端口号 7050 的设定，另外还有 ORDERER_GENERAL_GENESISFILE 、 ORDERER_GENERAL_LOCALMSPID 以及 ORDERER_GENERAL_LOCALMSPDIR 的参数需要确保与生成证书文件中所采用的配置文件上下文一致，且其中指定的文件映射路径需要与当前配置文件中 volumes 卷宗指向匹配，以免在启动的时候出现找不到创世区块等异常。

## 4.3 部署peer0.org1节点

有了 Orderer 排序服务启动文件，还需要专门为 Peer 节点所准备的 docker-peer.yaml 启动文件，Orderer 和 Peer 的启动 yaml 文件名称可根据实际需求自定义，具体内部源码如下：

```
version: '2'

services:

  couchdb:
    container_name: couchdb
    image: hyperledger/fabric-couchdb
```

```
      # Comment/Uncomment the port mapping if you want to hide/expose the CouchDB
service,
      # for example map it to utilize Fauxton User Interface in dev environments.
      ports:
        - "5984:5984"

    ca:
      container_name: ca
      image: hyperledger/fabric-ca
      environment:
        - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
        - FABRIC_CA_SERVER_CA_NAME=ca
        - FABRIC_CA_SERVER_TLS_ENABLED=false
        - FABRIC_CA_SERVER_TLS_CERTFILE=/etc/hyperledger/fabric-ca-server-
config/ca.org1.example.com-cert.pem
        - FABRIC_CA_SERVER_TLS_KEYFILE=/etc/hyperledger/fabric-ca-server-
config/95e05e630b6fd2f16b6367823c3a1295cc86e96431dd87b1376bea1d6120eb90_sk
      ports:
        - "7054:7054"
      command: sh -c 'fabric-ca-server start --ca.certfile /etc/hyperledger/fabric-
ca-server-config/ca.org1.example.com-cert.pem --ca.keyfile /etc/hyperledger/fabric-ca-
server-config/95e05e630b6fd2f16b6367823c3a1295cc86e96431dd87b1376bea1d6120eb90_sk -b
admin:adminpw -d'
      volumes:
        - ./crypto-config/peerOrganizations/org1.example.com/ca/:/etc/hyperledger/
fabric-ca-server-config

    peer0.org1.example.com:
      container_name: peer0.org1.example.com
      image: hyperledger/fabric-peer
      environment:
        - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
        - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb:5984

        - CORE_PEER_ID=peer0.org1.example.com
        - CORE_PEER_NETWORKID=aberic
        - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
        - CORE_PEER_CHAINCODELISTENADDRESS=peer0.org1.example.com:7052
        - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051
        - CORE_PEER_LOCALMSPID=Org1MSP

        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # the following setting starts chaincode containers on the same
        # bridge network as the peers
```

```
        # https://docs.docker.com/compose/networking/
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
        - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
        - CORE_PEER_GOSSIP_USELEADERELECTION=true
        - CORE_PEER_GOSSIP_ORGLEADER=false
        - CORE_PEER_PROFILE_ENABLED=false
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
      volumes:
          - /var/run/:/host/var/run/
          - ./crypto-config/peerOrganizations/org1.example.com/peers/peer0.
org1.example.com/msp:/etc/hyperledger/fabric/msp
          - ./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.
example.com/tls:/etc/hyperledger/fabric/tls
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      command: peer node start
      ports:
        - 7051:7051
        - 7052:7052
        - 7053:7053
      depends_on:
        - couchdb
      networks:
        default:
          aliases:
            - aberic

    cli:
      container_name: cli
      image: hyperledger/fabric-tools
      tty: true
      environment:
        - GOPATH=/opt/gopath
        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_PEER_ID=cli
        - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
        - CORE_PEER_LOCALMSPID=Org1MSP
```

```
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric
/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.cr
t
        - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      volumes:
        - /var/run/:/host/var/run/
        -
 ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/aberic/chaincode/go
        - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
        - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer
/channel-artifacts
      depends_on:
        - peer0.org1.example.com
```

Peer 的启动文件包含内容较多，比官方 e2e_cli 案例中的要稍全面一些，里面有 cli 客户端、CouchDB 插件以及 CA 服务端插件。

这里有几个地方需要注意修改，首先是 CA 部分有两处，一处是 FABRIC_CA_SERVER_TLS_KEYFILE，另一处是 command 中最后一部分。这两处的_sk 文件名称需要替换成之前生成的证书文件名称，其实这里的主要目的是加载 CA 并登录 CA 用户。

Peer 启动文件启动的是 peer0.org1.example.com 节点，所以对应的 CA 证书文件在 peer0.org1.example.com 下可以找到，具体路径是/home/docker/github.com/hyperledger/ fabric/aberic/crypto-config/peerOrganizations/org1.example.com/ca

目标如图 4-10 所示。

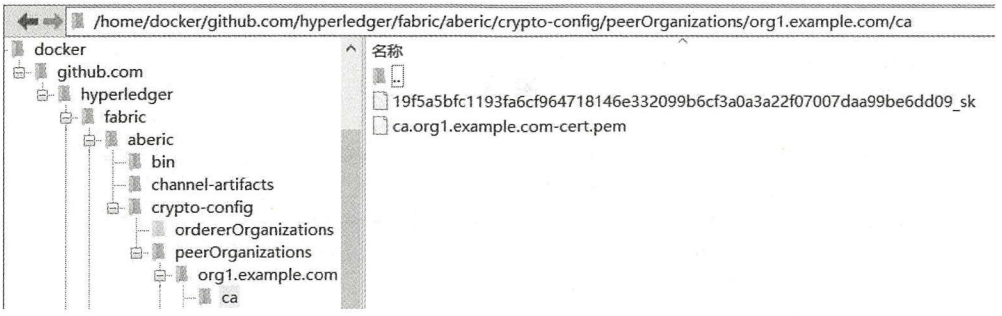


图4-10　CA文件

最后将 docker-peer.yaml 中两处的 95e05e630b6fd2f16b6367823c3a1295cc86e96431dd87b1376bea1d6120eb90_sk 替换成 19f5a5bfc1193fa6cf964718146e332099b6cf3a0a3a22f07007daa99be6dd09_sk 即可。

**注意：这里的替换是替换自己生成的，本书只是一个 Demo 说明，以免到时无法跑通。**

另外，本步骤中启动 Peer 节点可能会遇到类似如下错误：

```
fatal error: unexpected signal during runtime execution
[signal SIGSEGV: segmentation violation code=0x1 addr=0x63 pc=0x7ff619e21259]

runtime stack:
runtime.throw(0xf11259, 0x2a)
 /opt/go/src/runtime/panic.go:605 +0x95
runtime.sigpanic()
 /opt/go/src/runtime/signal_unix.go:351 +0x2b8

goroutine 41 [syscall, locked to thread]:
runtime.cgocall(0xbf3800, 0xc4201dfac8, 0xf0fa21)
 /opt/go/src/runtime/cgocall.go:132 +0xe4 fp=0xc4201dfa88 sp=0xc4201dfa48
pc=0x4023b4
net._C2func_getaddrinfo(0x7ff6040008c0, 0x0, 0xc42023a420, 0xc42000e5e8, 0x0, 0x0,
0x0)
 net/_obj/_cgo_gotypes.go:86 +0x5f fp=0xc4201dfac8 sp=0xc4201dfa88 pc=0x5f893f
net.cgoLookupIPCNAME.func2(0x7ff6040008c0, 0x0, 0xc42023a420, 0xc42000e5e8, 0x10,
0x6ed9f2, 0xc4201f0b78)
 /opt/go/src/net/cgo_unix.go:151 +0x13f fp=0xc4201dfb20 sp=0xc4201dfac8 pc=0x5ffedf
net.cgoLookupIPCNAME(0xc42001b634, 0x7, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0)
 /opt/go/src/net/cgo_unix.go:151 +0x175 fp=0xc4201dfc18 sp=0xc4201dfb20 pc=0x5fa195
net.cgoLookupIP(0x1667b00, 0xc42001a090, 0xc42001b634, 0x7, 0xc4201620a0,
0xc4201620c0, 0xc4201620e0, 0xc420163420, 0xc420163440, 0xc420163460)
 /opt/go/src/net/cgo_unix.go:209 +0x38f fp=0xc4201dfd80 sp=0xc4201dfc18 pc=0x5fad1f
net.(*Resolver).lookupIP(0x169d370, 0x1667b00, 0xc42001a090, 0xc42001b634, 0x7,
0xc4201635c0, 0xc4201635e0, 0xc420163600, 0xc42016e540, 0xc42016e560)
 /opt/go/src/net/lookup_unix.go:95 +0x12d fp=0xc4201dfdf8 sp=0xc4201dfd80
pc=0x5e59bd
net.(*Resolver).(net.lookupIP)-fm(0x1667b00, 0xc42001a090, 0xc42001b634, 0x7,
0xc42016e640, 0xc42016e660, 0xc42016e680, 0xc42016e6a0, 0xc42016e6c0)
 /opt/go/src/net/lookup.go:187 +0x56 fp=0xc4201dfe58 sp=0xc4201dfdf8 pc=0x602836
net.glob..func10(0x1667b00, 0xc42001a090, 0xc42013ef40, 0xc42001b634, 0x7,
0xc42016e7e0, 0xc42016e800, 0xc42016e820, 0xc42016e840, 0xc42016e860)
 /opt/go/src/net/hook.go:19 +0x52 fp=0xc4201dfeb0 sp=0xc4201dfe58 pc=0x5fc722
net.(*Resolver).LookupIPAddr.func1(0xc42011e000, 0x1667b00, 0xc42001a090, 0x0)
```

```
    /opt/go/src/net/lookup.go:193 +0x5c fp=0xc4201dff38 sp=0xc4201dfeb0 pc=0x5febac
   internal/singleflight.(*Group).doCall(0x169d360, 0xc42013d680, 0xc42001b634, 0x7,
0xc42023a390)
    /opt/go/src/internal/singleflight/singleflight.go:93 +0x2e fp=0xc4201dffb8
sp=0xc4201dff38 pc=0x5c0dce
   runtime.goexit()
    /opt/go/src/runtime/asm_amd64.s:2337 +0x1 fp=0xc4201dffc0 sp=0xc4201dffb8
pc=0x45e391
   created by internal/singleflight.(*Group).DoChan
    /opt/go/src/internal/singleflight/singleflight.go:86 +0x31f
```

参考 https://yq.aliyun.com/articles/238940 文章，提供如下解决方案。

```
vim /etc/resolv.conf
```

将“options timeout:2 attempts:3 rotate single-request-reopen”这一行内容注释掉，随后再次尝试重新启动该节点。

## 4.4　搭建Fabric网络

启动 Fabric 网络之前需要将智能合约及在第 4.2 节和第 4.3 节中创建的 docker-orderer.yaml 和 docker-peer.yaml 文件上传至 aberic 目录下。

而在 docker-peer.yaml 文件中的 cli 客户端配置中指定了智能合约的部署路径，在 aberic 目录下创建与之对应的 chaincode 文件夹，并在该文件夹下创建 go 文件夹，表示合约目录下以 Go 语言为基础的合约目录。

把官方 Demo 中的 chaincode_example02 示例也一并上传到 go 目录下，之后会以该合约为基础进行测试。

最终结果显示如图 4-11 所示。

/home/docker/github.com/hyperledger/fabric/aberic

| 名称 | 大小 | 类型 | 修改时间 | 属性 | 所有者 |
|---|---|---|---|---|---|
| .. | | | | | |
| bin | | 文件夹 | 2018/3/20, 17:48 | drwxr-... | root |
| chaincode | | 文件夹 | 2018/4/10, 18:55 | drwxr-... | root |
| channel-artifacts | | 文件夹 | 2018/4/10, 10:43 | drwxr-... | root |
| crypto-config | | 文件夹 | 2018/4/10, 9:47 | drwxr-... | root |
| configtx.yaml | 5KB | YAML ... | 2018/3/20, 16:37 | -rw-r--r-- | root |
| crypto-config.yaml | 4KB | YAML ... | 2018/3/20, 16:37 | -rw-r--r-- | root |
| docker-orderer.yaml | 2KB | YAML ... | 2018/4/10, 18:55 | -rw-r--r-- | root |
| docker-peer.yaml | 5KB | YAML ... | 2018/4/10, 18:55 | -rw-r--r-- | root |

图4-11　启动目录结构

在服务器命令行中分别执行如下命令启动 orderer 和 peer，按照顺序应该先启动排序服务，最终完成结果如图 4-12 所示。

```
docker-compose -f docker-orderer.yaml up -d
docker-compose -f docker-peer.yaml up -d
```

```
[root@VM_139_63_centos aberic]# docker-compose -f docker-orderer.yaml up -d
Creating network "aberic_default" with the default driver
Creating orderer.example.com ... done
[root@VM_139_63_centos aberic]#
Creating couchdb ... done
Creating peer0.org1.example.com ... done
Creating cli ... done
Creating couchdb ...
Creating peer0.org1.example.com ...
Creating cli ...
[root@VM_139_63_centos aberic]#
```

图4-12　启动Fabric网络

如果进行到本步骤还会出现提示镜像未下载或加载镜像失败的问题，请参考本书第 2.4.2 节。

启动完成后，继续执行如下命令查看容器是否均已启动，具体结果如图 4-13 所示。

```
docker ps
```

```
[root@VM_139_63_centos aberic]# docker ps
CONTAINER ID        IMAGE                        COMMAND                  CREATED             STATUS
158f99d98c2e        hyperledger/fabric-tools     "/bin/bash"              41 minutes ago      Up 41 minutes
f4edfaa56d28        hyperledger/fabric-peer      "peer node start"        41 minutes ago      Up 41 minutes
7a9514a2df00        hyperledger/fabric-ca        "sh -c 'fabric-ca-se…"   41 minutes ago      Up 41 minutes
868d09b7b42d        hyperledger/fabric-couchdb   "tini -- /docker-ent…"   41 minutes ago      Up 41 minutes
00cf82623a46        hyperledger/fabric-orderer   "orderer"                41 minutes ago      Up 41 minutes
```

图4-13　网络启动

可以看到所有的容器都已经成功启动，接下来是对 Channel 和 Chaincode 执行操作了。

首先是 Channel 的创建及加盟操作。对 Peer 节点的操作基本都需要依赖客户端完成，本书到目前还没有用到第三方 SDK，但安装了 fabric-tools 镜像，即已启动容器服务中的 cli 客户端，可以通过如下命令进入客户端对 Channel 进行相关操作：

```
docker exec -it cli bash
```

上述命令是对 Docker 容器的常规操作，cli 则是 YAML 启动文件中定义的 container_name（容器名称），通过修改上述命令中的“cli”为其他容器名称，可以开启所写容器的内部服务。

随后，执行如下命令创建一个 channel：

```
peer channel create -o orderer.example.com:7050 -c mychannel -t 50 -f ./channel-
```

```
artifacts/mychannel.tx
```

**注意：**此 channel 的创建并非随意而为，而是之前创建过一个 mychannel.tx 文件，在创建该文件的时候就已经指定了 channelID 是 mychannel。

随后执行 ls 即可查看已经创建的 mychannel.block 文件。最终执行结果如图 4-14 所示。

```
root@334c9647747c:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer channel create -o orderer.example.com:7050 -c mychannel -t 50 -f ./channel-artifacts/mychannel.tx
2018-03-22 04:27:39.837 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-03-22 04:27:39.837 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-03-22 04:27:39.838 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and orderer connections initialized
2018-03-22 04:27:39.838 UTC [msp] GetLocalMSP -> DEBU 004 Returning existing local MSP
2018-03-22 04:27:39.838 UTC [msp] GetDefaultSigningIdentity -> DEBU 005 Obtaining default signing identity
2018-03-22 04:27:39.839 UTC [msp] GetLocalMSP -> DEBU 006 Returning existing local MSP
2018-03-22 04:27:39.839 UTC [msp] GetDefaultSigningIdentity -> DEBU 007 Obtaining default signing identity
2018-03-22 04:27:39.839 UTC [msp/identity] Sign -> DEBU 008 Sign: plaintext: 0AA2060A074F7267314D53501296062D...53616D706C65436F6E736F7274697560
2018-03-22 04:27:39.839 UTC [msp/identity] Sign -> DEBU 009 Sign: digest: 5AAD2DCF209768C41DCBC8851A0270BA6A0540E5D7F0461D456F7A3BB00D635E
2018-03-22 04:27:39.839 UTC [msp] GetLocalMSP -> DEBU 00a Returning existing local MSP
2018-03-22 04:27:39.839 UTC [msp] GetDefaultSigningIdentity -> DEBU 00b Obtaining default signing identity
2018-03-22 04:27:39.839 UTC [msp] GetLocalMSP -> DEBU 00c Returning existing local MSP
2018-03-22 04:27:39.839 UTC [msp] GetDefaultSigningIdentity -> DEBU 00d Obtaining default signing identity
2018-03-22 04:27:39.839 UTC [msp/identity] Sign -> DEBU 00e Sign: plaintext: 0AD9060A1508021A0608BBE1CCD50522...68B6D2799DD87EB86D2955C32071852F
2018-03-22 04:27:39.839 UTC [msp/identity] Sign -> DEBU 00f Sign: digest: D09CF7F25992C0239D624902AA07D6EC803A7C53EA9E0969F730B15A32CC7A10
2018-03-22 04:27:39.887 UTC [msp] GetLocalMSP -> DEBU 010 Returning existing local MSP
2018-03-22 04:27:39.887 UTC [msp] GetDefaultSigningIdentity -> DEBU 011 Obtaining default signing identity
2018-03-22 04:27:39.888 UTC [msp] GetLocalMSP -> DEBU 012 Returning existing local MSP
2018-03-22 04:27:39.888 UTC [msp] GetDefaultSigningIdentity -> DEBU 013 Obtaining default signing identity
2018-03-22 04:27:39.888 UTC [msp/identity] Sign -> DEBU 014 Sign: plaintext: 0AD9060A1508021A0608BBE1CCD50522...04030221D3A512080A021A0012021A00
2018-03-22 04:27:39.888 UTC [msp/identity] Sign -> DEBU 015 Sign: digest: 60A33C208156B86C6A322FD7FEDE34502458443A6526A60F259F438F2014F7FD
2018-03-22 04:27:39.895 UTC [channelCmd] readBlock -> DEBU 016 Got status: &{NOT_FOUND}
2018-03-22 04:27:39.895 UTC [msp] GetLocalMSP -> DEBU 017 Returning existing local MSP
2018-03-22 04:27:39.895 UTC [msp] GetDefaultSigningIdentity -> DEBU 018 Obtaining default signing identity
2018-03-22 04:27:39.896 UTC [channelCmd] InitCmdFactory -> INFO 019 Endorser and orderer connections initialized
2018-03-22 04:27:40.097 UTC [msp] GetLocalMSP -> DEBU 01a Returning existing local MSP
2018-03-22 04:27:40.097 UTC [msp] GetDefaultSigningIdentity -> DEBU 01b Obtaining default signing identity
2018-03-22 04:27:40.097 UTC [msp] GetLocalMSP -> DEBU 01c Returning existing local MSP
2018-03-22 04:27:40.097 UTC [msp] GetDefaultSigningIdentity -> DEBU 01d Obtaining default signing identity
2018-03-22 04:27:40.097 UTC [msp/identity] Sign -> DEBU 01e Sign: plaintext: 0AD9060A1508021A0608BCE1CCD50522...4BD73EDCB41A12080A021A0012021A00
2018-03-22 04:27:40.097 UTC [msp/identity] Sign -> DEBU 01f Sign: digest: 60E8CAB10B8F67575165EB2FC17F165B41828E94EADD1597B469A94C95243441
2018-03-22 04:27:40.102 UTC [channelCmd] readBlock -> DEBU 020 Received block: 0
2018-03-22 04:27:40.102 UTC [main] main -> INFO 021 Exiting.....
```

图4-14　创建频道

创建完 channel 后，需要执行如下命令，通过 mychannel.block 文件来加入该 channel，以便后续可以安装实例化并测试智能合约。具体结果如图 4-15 所示。

```
peer channel join -b mychannel.block
```

```
root@334c9647747c:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer channel join -b mychannel.block
2018-03-22 04:27:54.226 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-03-22 04:27:54.226 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-03-22 04:27:54.228 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and orderer connections initialized
2018-03-22 04:27:54.229 UTC [msp/identity] Sign -> DEBU 004 Sign: plaintext: 0A9F070A5B08011A0B08CAE1CCD50510...853CF8F3A5451A080A000A000A000A00
2018-03-22 04:27:54.229 UTC [msp/identity] Sign -> DEBU 005 Sign: digest: E7E26EC6F436FA5B39D278278C1141E46F11DBBC76433A499EE34BE5E47855DB
2018-03-22 04:27:54.423 UTC [channelCmd] executeJoin -> INFO 006 Successfully submitted proposal to join channel
2018-03-22 04:27:54.423 UTC [main] main -> INFO 007 Exiting.....
root@334c9647747c:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

图4-15　加入频道

至此，已经完成了 channel 的创建并成功加入了该 channel，即一个最小单位的 Fabric 网络已经成功搭建起来了。

## 4.5 初步接触智能合约

Fabric 网络搭建起来后，就需要在上面执行合适的智能合约来实现具体的功能，从而使得现实中的项目得以落地。

接下来便是智能合约的安装部署、实例化及功能测试，在之前上传了官方的智能合约 Demo 到 go 目录下，合约目录为/home/docker/github.com/hyperledger/fabric/aberic/chaincode/go/chaincode_example02，此目录也是即将安装的智能合约路径。

首先安装智能合约，执行如下命令。

```
    peer chaincode install -n mychannel -p github.com/hyperledger/fabric/aberic/
chaincode/go/chaincode_example02 -v 1.0
```

安装合约成功的日志如下所示：

```
    root@8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer
chaincode install -n mychannel -p github.com/hyperledger/fabric/aberic/chaincode/
go/chaincode_example02 -v 1.0
    2018-04-10 12:39:36.686 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local
MSP
    2018-04-10 12:39:36.686 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining
default signing identity
    2018-04-10 12:39:36.686 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003
Using default escc
    2018-04-10 12:39:36.686 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004
Using default vscc
    2018-04-10 12:39:36.686 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java
chaincode disabled
    2018-04-10 12:39:36.910 UTC [golang-platform] getCodeFromFS -> DEBU 006
getCodeFromFS github.com/hyperledger/fabric/aberic/chaincode/go/chaincode_example02
    2018-04-10 12:39:37.274 UTC [golang-platform] func1 -> DEBU 007 Discarding GOROOT
package fmt
    2018-04-10 12:39:37.274 UTC [golang-platform] func1 -> DEBU 008 Discarding
provided package github.com/hyperledger/fabric/core/chaincode/shim
    2018-04-10 12:39:37.274 UTC [golang-platform] func1 -> DEBU 009 Discarding
provided package github.com/hyperledger/fabric/protos/peer
    2018-04-10 12:39:37.274 UTC [golang-platform] func1 -> DEBU 00a Discarding GOROOT
package strconv
    2018-04-10 12:39:37.274 UTC [golang-platform] GetDeploymentPayload -> DEBU 00b
done
    2018-04-10 12:39:37.274 UTC [container] WriteFileToPackage -> DEBU 00c Writing
file to tarball: src/github.com/hyperledger/fabric/aberic/chaincode/go/chaincode
_example02/chaincode_example02.go
    2018-04-10 12:39:37.278 UTC [container] WriteFileToPackage -> DEBU 00d Writing
```

```
file to tarball:
src/github.com/hyperledger/fabric/aberic/chaincode/go/chaincode_example02/chaincode_ex
ample02_test.go
    2018-04-10 12:39:37.279 UTC [msp/identity] Sign -> DEBU 00e Sign: plaintext:
0A9C070A5C08031A0C0889E1B2D60510...8D3EFF0D0000FFFFC99F84FA00300000
    2018-04-10 12:39:37.279 UTC [msp/identity] Sign -> DEBU 00f Sign: digest:
F8D28E1BFB7605CB02A118E59CDEEF092D52D6E588ACD255EED98E1E1061303A
    2018-04-10 12:39:37.312 UTC [chaincodeCmd] install -> DEBU 010 Installed remotely
response:<status:200 payload:"OK" >
    2018-04-10 12:39:37.312 UTC [main] main -> INFO 011 Exiting.....
```

安装完成后需要进行实例化 chaincode，执行如下命令：

```
    peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n mychannel -
c '{"Args":["init","A","10","B","10"]}' -P "OR ('Org1MSP.member')" -v 1.0
```

实例化合约成功日志如下所示：

```
    root@8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer
chaincode instantiate -o orderer.example.com:7050 -C mychannel -n mychannel -c
'{"Args":["init","A","10","B","10"]}' -P "OR ('Org1MSP.member')" -v 1.0
    2018-04-10 12:43:28.862 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local
MSP
    2018-04-10 12:43:28.862 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining
default signing identity
    2018-04-10 12:43:28.864 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003
Using default escc
    2018-04-10 12:43:28.864 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004
Using default vscc
    2018-04-10 12:43:28.864 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java
chaincode disabled
    2018-04-10 12:43:28.865 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext:
0AA7070A6708031A0C08F0E2B2D60510...314D53500A04657363630A0476736363
    2018-04-10 12:43:28.865 UTC [msp/identity] Sign -> DEBU 007 Sign: digest:
DDC0C322E770BFDBA9A5C15B3F2D13726407F4F6D33D4B346D4471B0F6939E55
    2018-04-10 12:43:55.586 UTC [msp/identity] Sign -> DEBU 008 Sign: plaintext:
0AA7070A6708031A0C08F0E2B2D60510...AF32D99FC35E021D2E28269A858A0117
    2018-04-10 12:43:55.586 UTC [msp/identity] Sign -> DEBU 009 Sign: digest:
FCA85DD3EBD3A28FC1FCD77CB01DBB8F6EF1E4DD005B9401EA3294D734F11E01
    2018-04-10 12:43:55.603 UTC [main] main -> INFO 00a Exiting.....
```

**注意：**在具体实例化智能合约的命令中通过-P 加入了背书方案，其中指明了仅有 Org1 的成员才具备背书能力。关于背书的具体讲解会放在本书后续章节中，这里提到是为了说明智能合约的 invoke 操作只有 Org1 执行才会成功。

通过具体的实例化智能合约命令，可以观察到其中的-c 参数指定了智能合约初始化时传入的参数内容，需要对比智能合约中的初始化方法来判断在该操作中发生了什么。具体的合约方法如下：

```
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
 fmt.Println("ex02 Init")
 _, args := stub.GetFunctionAndParameters()
 var A, B string    // Entities
 var Aval, Bval int // Asset holdings
 var err error

 if len(args) != 4 {
     return shim.Error("Incorrect number of arguments. Expecting 4")
 }

 // Initialize the chaincode
 A = args[0]
 Aval, err = strconv.Atoi(args[1])
 if err != nil {
     return shim.Error("Expecting integer value for asset holding")
 }
 B = args[2]
 Bval, err = strconv.Atoi(args[3])
 if err != nil {
     return shim.Error("Expecting integer value for asset holding")
 }
 fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)

 // Write the state to the ledger
 err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
 if err != nil {
     return shim.Error(err.Error())
 }

 err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
 if err != nil {
     return shim.Error(err.Error())
 }

 return shim.Success(nil)
}
```

阅读该方法，可以看出该方法会接收四个参数，而在**-c '{"Args":["init","A","10","B","10"]}'**命令中传入了五个参数，第一个参数为方法名，后四个参数为该方法将要接收的



参数内容。

所以，在 init 方法中接收的四个参数分别是“A”、“10”、“B”和“10”。通过代码不难观察出初始化方法的目的是创建一个 key 为 A 的账户并给该账户一个值为 10 的资产，同时创建一个 key 为 B 的账户并也给该账户一个值为 10 的资产。

接下来需要验证该智能合约中的 invoke 和 query 方法。先看 query 方法，如下所示：

```
   func (t *SimpleChaincode) query(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
    var A string // Entities
    var err error

    if len(args) != 1 {
        return shim.Error("Incorrect number of arguments. Expecting name of the
person to query")
    }

    A = args[0]

    // Get the state from the ledger
    Avalbytes, err := stub.GetState(A)
    if err != nil {
        jsonResp := "{\"Error\":\"Failed to get state for " + A + "\"}"
        return shim.Error(jsonResp)
    }

    if Avalbytes == nil {
        jsonResp := "{\"Error\":\"Nil amount for " + A + "\"}"
        return shim.Error(jsonResp)
    }

    jsonResp := "{\"Name\":\"" + A + "\",\"Amount\":\"" + string(Avalbytes) + "\"}"
    fmt.Printf("Query Response:%s\n", jsonResp)
    return shim.Success(Avalbytes)
   }
```

该方法需要接收且仅能接受一个参数，所接收的参数为智能合约中所创建账户的 key 值，返回内容为该账户下的资产，执行如下命令对 A 账户的资产进行查询：

```
   peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'
```

查询后日志结果如下所示：

```
   2018-04-11 01:46:31.791 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local
MSP
```

```
    2018-04-11 01:46:31.792 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining
default signing identity
    2018-04-11 01:46:31.792 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003
Using default escc
    2018-04-11 01:46:31.792 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004
Using default vscc
    2018-04-11 01:46:31.792 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java
chaincode disabled
    2018-04-11 01:46:31.792 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext:
0AAC070A6C08031A0C08F7D1B5D60510...6E6E656C1A0A0A0571756572790A0141
    2018-04-11 01:46:31.792 UTC [msp/identity] Sign -> DEBU 007 Sign: digest:
AE901EC2B82E19D8F623B1D98CAEEC48C315CD93FE7A1F74DB3187F0DF3B1C1C
    Query Result: 10
    2018-04-11 01:46:31.830 UTC [main] main -> INFO 008 Exiting.....
```

同样，如果对 B 账户执行如下命令进行查询，会得到 B 的资产也为 10。

```
    peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","B"]}'
```

查询完整结果如图 4-16 所示。

```
root@8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'
2018-04-11 01:46:31.791 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-04-11 01:46:31.792 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-04-11 01:46:31.792 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-04-11 01:46:31.792 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-04-11 01:46:31.792 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java chaincode disabled
2018-04-11 01:46:31.792 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AAC070A6C08031A0C08F7D1B5D60510...6E6E656C1A0A0A0571756572790A0141
2018-04-11 01:46:31.792 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: AE901EC2B82E19D8F623B1D98CAEEC48C315CD93FE7A1F74DB3187F0DF3B1C1C
Query Result: 10
2018-04-11 01:46:31.830 UTC [main] main -> INFO 008 Exiting.....
root@8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/peer#
root@8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","B"]}'
2018-04-11 01:48:45.625 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-04-11 01:48:45.625 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-04-11 01:48:45.625 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-04-11 01:48:45.625 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-04-11 01:48:45.625 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java chaincode disabled
2018-04-11 01:48:45.626 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AAC070A6C08031A0C08FDD2B5D60510...6E6E656C1A0A0A0571756572790A0142
2018-04-11 01:48:45.626 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: 36EE245646AA85A379A5FD76C3BE488F78C6E3732BBF208DF91F2898EF8CC17B
Query Result: 10
2018-04-11 01:48:45.642 UTC [main] main -> INFO 008 Exiting.....
root@8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

图4-16　查询结果

接下来进入 invoke 方法，具体代码示例如下：

```
    func (t *SimpleChaincode) invoke(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
    var A, B string    // Entities
    var Aval, Bval int // Asset holdings
    var X int          // Transaction value
    var err error

    if len(args) != 3 {
        return shim.Error("Incorrect number of arguments. Expecting 3")
```



```
}

A = args[0]
B = args[1]

// Get the state from the ledger
// TODO: will be nice to have a GetAllState call to ledger
Avalbytes, err := stub.GetState(A)
if err != nil {
    return shim.Error("Failed to get state")
}
if Avalbytes == nil {
    return shim.Error("Entity not found")
}
Aval, _ = strconv.Atoi(string(Avalbytes))

Bvalbytes, err := stub.GetState(B)
if err != nil {
    return shim.Error("Failed to get state")
}
if Bvalbytes == nil {
    return shim.Error("Entity not found")
}
Bval, _ = strconv.Atoi(string(Bvalbytes))

// Perform the execution
X, err = strconv.Atoi(args[2])
if err != nil {
    return shim.Error("Invalid transaction amount, expecting a integer value")
}
Aval = Aval - X
Bval = Bval + X
fmt.Printf("Aval = %d, Bval = %d\n", Aval, Bval)

// Write the state back to the ledger
err = stub.PutState(A, []byte(strconv.Itoa(Aval)))
if err != nil {
    return shim.Error(err.Error())
}

err = stub.PutState(B, []byte(strconv.Itoa(Bval)))
if err != nil {
    return shim.Error(err.Error())
}
```

```
    return shim.Success(nil)
   }
```

仔细阅读该方法，可以看到该方法需要传入三个参数，第一个和第二个参数分别是不同的账户名，第三个参数是资产值。方法的核心是第一个账户将自己资产中第三个参数值的资产转移到第二个账户名下。如 A 账户转移 5 个资产单位到 B 账户下，执行如下命令：

```
peer chaincode invoke -C mychannel -n mychannel -c '{"Args":["invoke", "A", "B",
"5"]}'
```

上述命令执行完成后继续通过查询方法，再次查询账户 A 和账户 B 的资产，最终得到结果如图 4-17 所示。

```
root@8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'
2018-04-11 01:55:53.424 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-04-11 01:55:53.424 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-04-11 01:55:53.424 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-04-11 01:55:53.424 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-04-11 01:55:53.424 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java chaincode disabled
2018-04-11 01:55:53.424 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AAC070A6C08031A0C08A9D6B5D60510...6E6E656C1A0A0A0571756572790A0141
2018-04-11 01:55:53.424 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: BCFC610DB763FC4D46BF89100119E8452348955F9D913C7B228944BE96EB13F6
Query Result: 5
2018-04-11 01:55:53.443 UTC [main] main -> INFO 008 Exiting.....
root@8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","B"]}'
2018-04-11 01:55:55.981 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-04-11 01:55:55.981 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-04-11 01:55:55.981 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-04-11 01:55:55.981 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-04-11 01:55:55.981 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java chaincode disabled
2018-04-11 01:55:55.981 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AAC070A6C08031A0C08ABD685D60510...6E6E656C1A0A0A0571756572790A0142
2018-04-11 01:55:55.981 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: A9CD20AB00E8CCB40F7891FE34BBE1DB257E520FBA7BAD82CC0B3C75BC8B406D
Query Result: 15
2018-04-11 01:55:55.996 UTC [main] main -> INFO 008 Exiting.....
root@8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

图4-17　查询结果

可以看到 A 的资产减少了 5，而 B 的资产增加了 5，得到的结果分别是 5 和 15。

至此，智能合约的测试暂告一段落，关于智能合约在后续章节会有单独介绍，接下来需要部署第二个节点加入该 Fabric 网络当中。

## 4.6　部署peer0.org2节点

现在还需要专门为 Org2 的 Peer 节点准备新的 docker-peer1.yaml 启动文件，具体内部源码如下：

```
version: '2'

services:

  peer0.org2.example.com:
    container_name: peer0.org2.example.com
```



```
    image: hyperledger/fabric-peer
    environment:
      - CORE_PEER_ID=peer0.org2.example.com
      - CORE_PEER_NETWORKID=aberic
      - CORE_PEER_ADDRESS=peer0.org2.example.com:7051
      - CORE_PEER_CHAINCODELISTENADDRESS=peer0.org2.example.com:7052
      - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org2.example.com:7051
      - CORE_PEER_LOCALMSPID=Org2MSP

      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      # the following setting starts chaincode containers on the same
      # bridge network as the peers
      # https://docs.docker.com/compose/networking/
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic
      # - CORE_LOGGING_LEVEL=ERROR
      - CORE_LOGGING_LEVEL=DEBUG
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
      - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
      - CORE_PEER_GOSSIP_USELEADERELECTION=true
      - CORE_PEER_GOSSIP_ORGLEADER=false
      - CORE_PEER_PROFILE_ENABLED=false
      - CORE_PEER_TLS_ENABLED=false
      - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
      - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
      - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
    volumes:
        - /var/run/:/host/var/run/
        - ./crypto-config/peerOrganizations/org2.example.com/peers/peer0.org2.
example.com/msp:/etc/hyperledger/fabric/msp
        - ./crypto-config/peerOrganizations/org2.example.com/peers/peer0.
org2.example.com/tls:/etc/hyperledger/fabric/tls
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
    command: peer node start
    ports:
      - 8051:7051
      - 8052:7052
      - 8053:7053
    networks:
      default:
        aliases:
          - aberic
```

由于是单机多节点，在一台服务器上启动多个节点，多个节点尽管属于同一端口，但对外映射在物理机上的端口会有所区别，如同上述源码中 Peer 节点、监听及事件端口映射到物理

机上由第 4.3 节中的 7051、7052、7053 更改为 8051、8052 和 8053。

有了第 4.3 节的理解，该步骤中的源码阅读会简单很多，这时候只需要将 docker-peer1.yaml 文件上传至 aberic 目录即可，最终目录结构如图 4-18 所示。

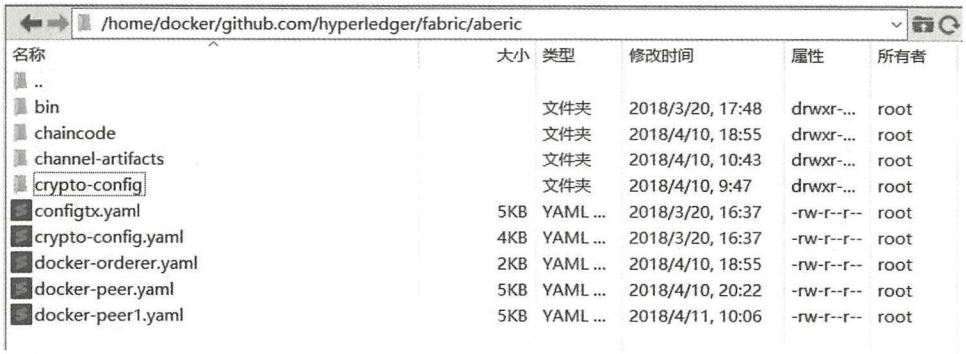


图4-18　项目结构图

进入服务器并进入/home/docker/github.com/hyperledger/fabric/aberic 项目目录下，执行如下命令启动 docker-peer1.yaml：

```
docker-compose -f docker-peer1.yaml up -d
```

最终得到如图 4-19 所示结果。

```
[root@VM_139_63_centos aberic]# docker-compose -f docker-peer1.yaml up -d
WARNING: Found orphan containers (cli, peer0.org1.example.com, ca, couchdb, orderer.example.com) for this project. If you removed
 --remove-orphans flag to clean it up.
Creating peer0.org2.example.com ... done
[root@VM_139_63_centos aberic]# docker ps
CONTAINER ID        IMAGE                                                
```

```
docker exec -it cli bash
```

随后，在进入该容器后进行一系列当前容器全局变量赋值操作，命令分别如下进行：

```
CORE_PEER_ID=peer0.org2.example.com
CORE_PEER_ADDRESS=peer0.org2.example.com:7051
CORE_PEER_CHAINCODELISTENADDRESS=peer0.org2.example.com:7052
CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org2.example.com:7051
CORE_PEER_LOCALMSPID=Org2MSP
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
```

cli 容器内的全局变量修改完成后，即可开始对 peer0org2 进行操作，与 peer0org1 一样，首先执行 peer0org2 加入频道的操作，具体执行如下命令：

```
peer channel join -b mychannel.block
peer chaincode install -n mychannel -p github.com/hyperledger/fabric/aberic/chaincode/go/chaincode_example02 -v 1.0
```

在 peer0org2 成功加入频道并安装智能合约后，可执行如下命令测试合约是否已成功运行：

```
peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'
```

最终会打印如下日志：

```
2018-04-11 06:47:37.617 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-04-11 06:47:37.617 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-04-11 06:47:37.617 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-04-11 06:47:37.617 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-04-11 06:47:37.617 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java chaincode disabled
2018-04-11 06:47:37.617 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AAC070A6C08031A0C0889DFB6D60510...6E6E656C1A0A0A0571756572790A0141
2018-04-11 06:47:37.617 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: 62D5E8C673BD7339759F6853BF937D7C75F75C790B0B733BBA49FD5AAC1C6D6B
Query Result: 5
2018-04-11 06:47:37.634 UTC [main] main -> INFO 008 Exiting.....
```

因为之前已经执行过一次账户资产变更的操作，所以现在查询出来 A 的资产是 5。此时，还是在当前 cli 容器中再次执行一次之前的命令，这一次让 B 给 A 转移价值 5 的资产，具体命

令如下所示：

```
   peer chaincode invoke -C mychannel -n mychannel -c '{"Args":["invoke", "B", "A",
"5"]}'
```

执行完成后再次对 A 的资产进行查询，这时会发现 A 的资产还是 5，没有发生变化，请参考本书第 4.5 节中对背书的注释内容。在实例化智能合约操作的时候，选择的背书组织为 Org1，而后加入的节点组织为 Org2，这说明来自 Org2 的对资产变更的操作都未经过背书，即不具备任何效力。但因为 Org2 组织加入了该频道且安装了合法的合约，可以对区块链中的数据进行检索。

执行 exit 退出当前 cli，再重新进入 cli 容器，这时 cli 容器默认的是对 peer0org1 节点的操作。执行之前的让 B 给 A 转移价值 5 的资产的命令，再继续执行一次查询操作，这时查到 A 的资产是 10，表示合约被执行成功，也应对了 Org1 具备背书能力的问题。

## 4.7 本章小结

本章带着读者通过纯手动的方式进行了一次 Fabric 网络最小单元的开发测试，这也为后续对多机多节点部署的理解打下了基础。

通过本章介绍，讲述了如何生成各节点证书及配置文件，生成创世区块和频道证书文件，也分析了生成这些文件所需配置文件的编写方案。同时，也通过 YAML 配置文件的编写启动了 Orderer 排序服务节点及 Peer 节点，并实现了一次伪动态加盟的操作。其中，对频道的新建、加入等操作有过介绍，对智能合约的基本概念及具体实施有实践。

每一个开始学习 HyperLedger Fabric 的开发人员都需要经历这样一个过程，从跑通第 3 章的 e2e_cli 案例到自己纯手动跑一遍整个 Fabric 网络，加深对 Fabric 的印象和理解。后续章节中的很多操作都会与本章类似甚至相同，且会将本章注释中的内容逐步拆分讲解，当阅读完后续章节后，再回头来看本章，会有全新的理解。

---

**注意：**本章中所有 YAML 文件源码及所生成的配置文件等均可在本书前言的“读者服务”中找到并进行下载。

---

# 第 5 章　Solo 多机部署

本章为第 4 章的扩展内容，第 4 章是单机实现的多节点部署方案，本章将采用多台物理服务器实现多机多节点部署。

部署之初所需要准备的各节点证书文件以及创世区块、频道加盟文件等均可沿用第 4 章中的内容，本章不再赘述。

## 5.1　网络拓扑

Solo 类型启动多机多节点部署需要至少两台服务器来完成最小单元的 Fabric 网络组网工作，每新增一台额外的服务器都可以作为新的 Peer 节点服务器加盟。本章因为沿用了第 4 章的配置文件，故需要准备 2~5 台服务器完成部署，具体拓扑如图 5-1 所示。

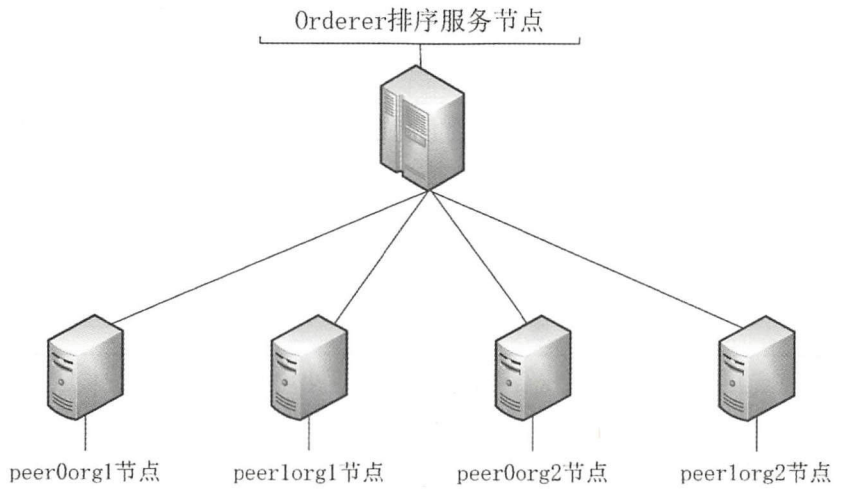


图5-1　多机网络

本书中的网络采用 5 台服务器测试 Fabric 组网工作，其中准备 4 台服务器用来运行 Peer 节点，另外 1 台用来运行 Orderer 排序服务节点。

注意：Hyperledger Fabric 网络中的 Peer 节点服务器对存储空间的消耗比较大，在实际生产体验的过程中，每一条请求数据大概仅 2K，但实际占用空间远不止这些。在第 3.3.1 节中提到过 MaxMessageCount 属性，其中也介绍了它的含义，合理的利用会避免存储膨胀等意外问题出现。

一般情况下，不会出现大批量数据导入的时候，节点服务器采用 8C8G1T 的配置足够用很久，而且 Fabric1.1 版本对 CouchDB 进行了优化，空间方面更加好用。

相对于内存来说，CPU 资源瓶颈要出现得早一点，这点官方也没有给出建议配置，只能自己摸索。

在一些文章中可以找到一些官方资料的线索，比如 Hyperledger Fabric 1.0 版本的目标是支持 1000TPS，且实验室数据已经达到 300~400TPS 了。这些文章没有明确到底是节点服务器还是排序服务器广播的值，在实际操作中，一台节点服务器大约在 200 TPS 以上，再往上没有尝试。

这里给出一条日常使用最低配置建议，配置见表 5-1。

表 5-1 日常使用最低配置

| 服务器类型 | 建议配置 | 备 注 |
| --- | --- | --- |
| Zookeeper | 4C+4G+500G | 磁盘有动态扩容需求 |
| Kafka | 4C+4G+500G | 磁盘有动态扩容需求 |
| Orderer | 8C+4G+500G | 磁盘有动态扩容需求 |
| Peer | 4C+4G+1T | 磁盘有动态扩容需求，有大数据导入提前扩容 CPU |

如果存在大批量数据导入的情况，Orderer 和 Peer 的配置建议 16C+16G 起步，以免在数据导入过程中容器宕机从而导致数据丢失的情况发生。

本书根据实际需求，启用了如下 5 台服务器，见表 5-2。

表 5-2 Solo 服务器部署

| 名 称 | IP | Hostname | 组织机构 |
| --- | --- | --- | --- |
| orderer | 172.31.159.130 | orderer.example.com | Orderer |
| peer0 | 172.31.143.22 | peer0.org1.example.com | Org1 |
| peer1 | 172.31.143.23 | peer1.org1.example.com | Org1 |
| peer0 | 172.31.159.129 | peer0.org2.example.com | Org2 |
| peer1 | 172.31.143.21 | peer1.org2.example.com | Org2 |

首先需要参考第 1 章和第 2 章的内容对 5 台服务器全部安装好 Fabric 的运行环境，并参照第 4 章的操作，在 Fabric 源码中新建 aberic 文件夹作为本次运行的项目目录。

如果有镜像功能，可以先做好 1 台服务器，然后直接镜像到其他 4 台服务器以节约部署时间。在已有第 4 章准备好的配置文件的情况下，与第 4 章的步骤一样，需要先行配置并启动 Orderer 排序服务节点。

## 5.2　部署Orderer节点

配置 docker-orderer.yaml 启动文件，具体源码如下所示：

```
version: '2'

services:

  orderer.example.com:
    container_name: orderer.example.com
    image: hyperledger/fabric-orderer
    environment:
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
      # - ORDERER_GENERAL_LOGLEVEL=error
      - ORDERER_GENERAL_LOGLEVEL=debug
      - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
      - ORDERER_GENERAL_LISTENPORT=7050
      #- ORDERER_GENERAL_GENESISPROFILE=AntiMothOrdererGenesis
      - ORDERER_GENERAL_GENESISMETHOD=file
      - ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block
      - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
      - ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
      #- ORDERER_GENERAL_LEDGERTYPE=ram
      #- ORDERER_GENERAL_LEDGERTYPE=file
      # enabled TLS
      - ORDERER_GENERAL_TLS_ENABLED=false
      - ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
      - ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
      - ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric
    command: orderer
    volumes:
    - ./channel-artifacts/genesis.block:/var/hyperledger/orderer/orderer.genesis.
block
    - ./crypto-config/ordererOrganizations/example.com/orderers/orderer.
example.com/msp:/var/hyperledger/orderer/msp
    - ./crypto-config/ordererOrganizations/example.com/orderers/orderer.example.
com/tls/:/var/hyperledger/orderer/tls
```

```
    networks:
      default:
        aliases:
          - aberic
    ports:
      - 7050:7050
```

本次配置文件与第 4.2 节基本一样，在该配置中的 LogLevel 设置为 DeBug，以便在其他加盟的 Peer 节点出现问题时可以查看 Orderer 中反馈的日志信息。

Orderer 排序服务节点的启动也比较简单，将 docker-orderer.yaml 文件上传至 172.31.159.130 服务器的/home/docker/github.com/hyperledger/fabric/aberic 目录下，并运行如下命令启动 Orderer 排序服务节点：

```
docker-compose -f docker-orderer.yaml up -d
```

启动完成后，再次通过如下命令查看是否启动成功：

```
docker ps
```

随后开始进一步处理 Peer 节点。

## 5.3 部署peer0.org1节点

Orderer 排序服务节点单独启动后，为 Org1 组织 Peer0 节点所准备的 docker-peer01.yaml 启动文件，其源码如下：

```
version: '2'

services:

  couchdb:
    container_name: couchdb
    image: hyperledger/fabric-couchdb
    # Comment/Uncomment the port mapping if you want to hide/expose the CouchDB
service,
    # for example map it to utilize Fauxton User Interface in dev environments.
    ports:
      - "5984:5984"

  ca:
    container_name: ca
    image: hyperledger/fabric-ca
    environment:
```

```
        - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
        - FABRIC_CA_SERVER_CA_NAME=ca
        - FABRIC_CA_SERVER_TLS_ENABLED=false
        - FABRIC_CA_SERVER_TLS_CERTFILE=/etc/hyperledger/fabric-ca-server-config/ca.
org1.example.com-cert.pem
        - FABRIC_CA_SERVER_TLS_KEYFILE=/etc/hyperledger/fabric-ca-server-
config/ab7dca5e5f6b1cc24c2023764c5b34d1f78d8614d2a11e74178d2d5509bd3be8_sk
      ports:
        - "7054:7054"
      command: sh -c 'fabric-ca-server start --ca.certfile /etc/hyperledger/fabric-
ca-server-config/ca.org1.example.com-cert.pem --ca.keyfile /etc/hyperledger/fabric-ca-
server-config/ab7dca5e5f6b1cc24c2023764c5b34d1f78d8614d2a11e74178d2d5509bd3be8_sk -b
admin:adminpw -d'
      volumes:
        - ./crypto-config/peerOrganizations/org1.example.com/ca/:/etc/hyperledger/
fabric-ca-server-config

    peer0.org1.example.com:
      container_name: peer0.org1.example.com
      image: hyperledger/fabric-peer
      environment:
        - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
        - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb:5984

        - CORE_PEER_ID=peer0.org1.example.com
        - CORE_PEER_NETWORKID=aberic
        - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
        - CORE_PEER_CHAINCODELISTENADDRESS=peer0.org1.example.com:7052
        - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051
        - CORE_PEER_LOCALMSPID=Org1MSP

        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # the following setting starts chaincode containers on the same
        # bridge network as the peers
        # https://docs.docker.com/compose/networking/
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
        - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
        - CORE_PEER_GOSSIP_USELEADERELECTION=true
        - CORE_PEER_GOSSIP_ORGLEADER=false
        - CORE_PEER_PROFILE_ENABLED=false
        - CORE_PEER_TLS_ENABLED=false
```

```
        - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
      volumes:
          - /var/run/:/host/var/run/
          - ./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.
example.com/msp:/etc/hyperledger/fabric/msp
          - ./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.
example.com/tls:/etc/hyperledger/fabric/tls
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      command: peer node start
      ports:
        - 7051:7051
        - 7052:7052
        - 7053:7053
      depends_on:
        - couchdb
      networks:
        default:
          aliases:
            - aberic
      extra_hosts:
       - "orderer.example.com:172.31.159.130"

    cli:
      container_name: cli
      image: hyperledger/fabric-tools
      tty: true
      environment:
        - GOPATH=/opt/gopath
        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_PEER_ID=cli
        - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
        - CORE_PEER_LOCALMSPID=Org1MSP
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
        - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
```

```
crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
        working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
        volumes:
            - /var/run/:/host/var/run/
            - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/aberic/
chaincode/go
            - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
            - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/
channel-artifacts
        depends_on:
          - peer0.org1.example.com
        extra_hosts:
         - "orderer.example.com:172.31.159.130"
         - "peer0.org1.example.com:172.31.143.22"
```

其中注意事项参考第 4.3 节，随后将 docker-peer01.yaml 文件上传至 172.31.143.22 服务器 /home/docker/github.com/hyperledger/fabric/aberic 目录下，并执行如下命令启动该节点：

```
docker-compose -f docker-peer01.yaml up -d
```

启动完成后开始进行第 4.4 节和第 4.5 节中介绍的组网过程。

通过如下命令进入客户端进行对 Channel 的相关操作：

```
docker exec -it cli bash
```

随后执行如下命令创建一个 channel：

```
peer channel create -o orderer.example.com:7050 -c mychannel -t 50 -f ./channel-
artifacts/mychannel.tx
```

创建完成后，执行如下命令，通过 mychannel.block 文件加入该 channel，以便后续可以安装实例化并测试智能合约。

```
peer channel join -b mychannel.block
```

安装智能合约，执行如下命令：

```
peer chaincode install -n mychannel -p github.com/hyperledger/fabric/aberic/
chaincode/go/chaincode_example02 -v 1.0
```

安装完成后进行实例化 chaincode，执行如下命令：

```
peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n mychannel -
c '{"Args":["init","A","10","B","10"]}' -P "OR ('Org1MSP.member')" -v 1.0
```

执行如下命令，对智能合约中 A 账户的资产进行查询：

```
peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'
```

结果如图 5-2 所示。

```
root@998026d4ae7b:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'
2018-04-12 12:53:42.579 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-04-12 12:53:42.579 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-04-12 12:53:42.579 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-04-12 12:53:42.579 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-04-12 12:53:42.579 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java chaincode disabled
2018-04-12 12:53:42.579 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AAC070A6C08031A0C08D6ADBDD60510...6E6E656C1A0A0A0571756572790A0141
2018-04-12 12:53:42.579 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: E714E8A4D1C59AF3692A08B1482095D870700997536287 6A103539C24D969123
Query Result: 10
2018-04-12 12:53:42.600 UTC [main] main -> INFO 008 Exiting.....
root@998026d4ae7b:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

图5-2　查询资产

接着，执行如下命令对智能合约中 B 账户的资产进行查询：

```
peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","B"]}'
```

查询结果 B 的资产也为 10。

继续测试智能合约并将 A 账户转移 5 个资产单位到 B 账户下，执行如下命令：

```
peer chaincode invoke -C mychannel -n mychannel -c '{"Args":["invoke", "A", "B",
"5"]}'
```

随后分别查询 A 资产和 B 资产，结果如图 5-3 所示。

```
root@998026d4ae7b:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'
2018-04-12 12:57:04.728 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-04-12 12:57:04.728 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-04-12 12:57:04.729 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-04-12 12:57:04.729 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-04-12 12:57:04.729 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java chaincode disabled
2018-04-12 12:57:04.729 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AAC070A6C08031A0C08A0AFBDD60510...6E6E656C1A0A0A0571756572790A0141
2018-04-12 12:57:04.729 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: 86A409405F1E9A9F22A43234995BB9444D54622AFB490B81B34E7DFEA778DB47
Query Result: 5
2018-04-12 12:57:04.751 UTC [main] main -> INFO 008 Exiting.....
root@998026d4ae7b:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","B"]}'
2018-04-12 12:57:09.874 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-04-12 12:57:09.874 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-04-12 12:57:09.874 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-04-12 12:57:09.874 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-04-12 12:57:09.874 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java chaincode disabled
2018-04-12 12:57:09.874 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AAC070A6C08031A0C08A5AFBDD60510...6E6E656C1A0A0A0571756572790A0142
2018-04-12 12:57:09.874 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: 06363550020AB7BAD71854BAB18DC0025B539BE9B4CE96EE25E4FDABEA98F5D6
Query Result: 15
2018-04-12 12:57:09.887 UTC [main] main -> INFO 008 Exiting.....
root@998026d4ae7b:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

图5-3　A资产和B资产查询

结果如 invoke 操作一致，A 的资产变为 5，B 的资产变为 15，经过智能合约操作后实现了资产转移。

至此，peer0.org1 已经部署完成，接下来继续将余下的节点一台一台地进行部署。

## 5.4 部署peer1.org1节点

下面开始部署 peer1.org1 节点，仍然需要准备该节点的配置文件，命名为 docker-peer11.yaml，具体源码如下：

```
    version: '2'

    services:

      couchdb:
        container_name: couchdb
        image: hyperledger/fabric-couchdb
        # Comment/Uncomment the port mapping if you want to hide/expose the CouchDB
service,
        # for example map it to utilize Fauxton User Interface in dev environments.
        ports:
          - "5984:5984"

      ca:
        container_name: ca
        image: hyperledger/fabric-ca
        environment:
          - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
          - FABRIC_CA_SERVER_CA_NAME=ca
          - FABRIC_CA_SERVER_TLS_ENABLED=false
          - FABRIC_CA_SERVER_TLS_CERTFILE=/etc/hyperledger/fabric-ca-server-
config/ca.org1.example.com-cert.pem
          - FABRIC_CA_SERVER_TLS_KEYFILE=/etc/hyperledger/fabric-ca-server-
config/ab7dca5e5f6b1cc24c2023764c5b34d1f78d8614d2a11e74178d2d5509bd3be8_sk
        ports:
          - "7054:7054"
        command: sh -c 'fabric-ca-server start --ca.certfile /etc/hyperledger/fabric-
ca-server-config/ca.org1.example.com-cert.pem --ca.keyfile /etc/hyperledger/fabric-ca-
server-config/ab7dca5e5f6b1cc24c2023764c5b34d1f78d8614d2a11e74178d2d5509bd3be8_sk -b
admin:adminpw -d'
        volumes:
          - ./crypto-
config/peerOrganizations/org1.example.com/ca/:/etc/hyperledger/fabric-ca-server-config

      peer1.org1.example.com:
        container_name: peer1.org1.example.com
        image: hyperledger/fabric-peer
        environment:
          - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
```

```
      - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb:5984

      - CORE_PEER_ID=peer1.org1.example.com
      - CORE_PEER_NETWORKID=aberic
      - CORE_PEER_ADDRESS=peer1.org1.example.com:7051
      - CORE_PEER_CHAINCODELISTENADDRESS=peer1.org1.example.com:7052
      - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer1.org1.example.com:7051
      - CORE_PEER_LOCALMSPID=Org1MSP

      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      # the following setting starts chaincode containers on the same
      # bridge network as the peers
      # https://docs.docker.com/compose/networking/
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic
      # - CORE_LOGGING_LEVEL=ERROR
      - CORE_LOGGING_LEVEL=DEBUG
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
      - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
      - CORE_PEER_GOSSIP_USELEADERELECTION=true
      - CORE_PEER_GOSSIP_ORGLEADER=false
      - CORE_PEER_PROFILE_ENABLED=false
      - CORE_PEER_TLS_ENABLED=false
      - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
      - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
      - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
    volumes:
        - /var/run/:/host/var/run/
        - ./crypto-config/peerOrganizations/org1.example.com/peers/peer1.org1.
example.com/msp:/etc/hyperledger/fabric/msp
        - ./crypto-
config/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls:/etc/hyperl
edger/fabric/tls
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
    command: peer node start
    ports:
      - 7051:7051
      - 7052:7052
      - 7053:7053
    depends_on:
      - couchdb
    networks:
      default:
        aliases:
          - aberic
```

```
      extra_hosts:
       - "orderer.example.com:172.31.159.130"

    cli:
      container_name: cli
      image: hyperledger/fabric-tools
      tty: true
      environment:
        - GOPATH=/opt/gopath
        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_PEER_ID=cli
        - CORE_PEER_ADDRESS=peer1.org1.example.com:7051
        - CORE_PEER_LOCALMSPID=Org1MSP
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls/ca.crt
        - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      volumes:
          - /var/run/:/host/var/run/
          - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/aberic/
chaincode/go
          - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
          - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/
channel-artifacts
      depends_on:
        - peer1.org1.example.com
      extra_hosts:
       - "orderer.example.com:172.31.159.130"
       - "peer1.org1.example.com:172.31.143.23"
```

将新建的 docker-peer11.yaml 上传至 172.31.143.23 服务器/home/docker/github.com/hyperledger/fabric/aberic 目录下，执行如下命令启动 peer1.org1：

```
docker-compose -f docker-peer11.yaml up -d
```

peer1.org1 启动完成后，后续操作主要是加入已有频道、安装并测试智能合约。

首先是加入频道，peer1org1 若想要加入某一指定频道，需要获取该频道的 block 文件，并以此为入口加入。

在第 5.3 节中，通过 peer0org1 创建一个 mychannel.block 文件，该文件位于 cli 容器内 /opt/gopath/src/github.com/hyperledger/fabric/peer 目录下，进入该容器并查看容器下文件的内容，结果如图 5-4 所示。

```
[root@VM_139_63_centos aberic]# docker exec -it cli bash
root@8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/peer# ls
channel-artifacts  crypto  mychannel.block
root@8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

图5-4　cli容器

通过图 5-4 所示，在 cli 容器内/opt/gopath/src/github.com/hyperledger/fabric/peer 目录下可以看到 mychannel.block 文件，现在需要将该文件从容器中复制到物理机真实路径下，退出当前容器并执行如下命令：

```
docker cp 8063b37180d8:/opt/gopath/src/github.com/hyperledger/fabric/
peer/mychannel.block /home/docker/github.com/hyperledger/fabric/aberic/channel-
artifacts/
```

命令中的 8063b37180d8 为 cli 容器的 CONTAINER ID，意思是通过 docker cp 命令将 cli 容器/opt/gopath/src/github.com/hyperledger/fabric/peer 目录下的 mychannel.block 文件复制到物理机/home/docker/github.com/hyperledger/fabric/aberic/channel-artifacts 目录下，最终复制出来后可通过 FTP 看到如图 5-5 所示目录结构。


图5-5　mychannle.block文件

此时，需要将 mychannel.block 文件再次从物理机复制到 peer1org1 的客户端中，以便通过客户端执行 Peer 节点加入频道，具体执行如下命令：

```
docker cp /home/docker/github.com/hyperledger/fabric/aberic/channel-artifacts/
mychannel.block aa2f0b716c42:/opt/gopath/src/github.com/hyperledger/fabric/peer/
```

命令中的 aa2f0b716c42 为 cli1 容器的 CONTAINER ID，意思是通过 docker cp 命令将物理机/home/docker/github.com/hyperledger/fabric/aberic/channel-artifacts 目录下的 mychannel.block

文件复制到 cli 容器/opt/gopath/src/github.com/hyperledger/fabric/peer 目录下，最终复制出来后可看到如图 5-6 所示结果。

```
[root@VM_139_63_centos aberic]# docker exec -it cli1 bash
root@aa2f0b716c42:/opt/gopath/src/github.com/hyperledger/fabric/peer# ls
channel-artifacts  crypto  mychannel.block
root@aa2f0b716c42:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

图5-6　cli容器

通过图 5-6 所示，在 cli 容器内/opt/gopath/src/github.com/hyperledger/fabric/peer 目录下可以看到 mychannel.block 文件，可以执行 peer1org1 加入频道的操作了，具体执行如下命令：

```
peer channel join -b mychannel.block
```

安装智能合约，执行如下命令：

```
peer chaincode install -n mychannel -p github.com/hyperledger/fabric/aberic/
chaincode/go/chaincode_example02 -v 1.0
```

安装完成后无需再次进行实例化 chaincode，直接执行如下命令，对智能合约中 A 账户的资产进行查询：

```
peer chaincode query -C mychannel -n mychannel -c '{"Args":["query","A"]}'
```

会得到与第 5.3 节后续操作中一样的结果，并继续按照第 5.3 节中后续对智能合约的操作进行一次简单测试，这里不再赘述。

## 5.5　部署peer0.org2节点

部署 peer0.org2 节点，准备该节点的配置文件，命名为 docker-peer02.yaml，具体源码如下：

```
version: '2'

services:

  couchdb:
    container_name: couchdb
    image: hyperledger/fabric-couchdb
    # Comment/Uncomment the port mapping if you want to hide/expose the CouchDB
service,
    # for example map it to utilize Fauxton User Interface in dev environments.
    ports:
      - "5984:5984"
```

```
    ca:
      container_name: ca
      image: hyperledger/fabric-ca
      environment:
        - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
        - FABRIC_CA_SERVER_CA_NAME=ca
        - FABRIC_CA_SERVER_TLS_ENABLED=false
        - FABRIC_CA_SERVER_TLS_CERTFILE=/etc/hyperledger/fabric-ca-server-config/
ca.org2.example.com-cert.pem
        - FABRIC_CA_SERVER_TLS_KEYFILE=/etc/hyperledger/fabric-ca-server-config/
638c62f6073cf84d4a897b90fa31c0d53c4c2ffaa0b0be1c3c7df233eb33a27a_sk
      ports:
        - "7054:7054"
      command: sh -c 'fabric-ca-server start --ca.certfile /etc/hyperledger/fabric-
ca-server-config/ca.org2.example.com-cert.pem --ca.keyfile /etc/hyperledger/fabric-ca-
server-config/638c62f6073cf84d4a897b90fa31c0d53c4c2ffaa0b0be1c3c7df233eb33a27a_sk -b
admin:adminpw -d'
      volumes:
        - ./crypto-config/peerOrganizations/org2.example.com/ca/:/etc/hyperledger/
fabric-ca-server-config

    peer0.org2.example.com:
      container_name: peer0.org2.example.com
      image: hyperledger/fabric-peer
      environment:
        - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
        - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb:5984

        - CORE_PEER_ID=peer0.org2.example.com
        - CORE_PEER_NETWORKID=aberic
        - CORE_PEER_ADDRESS=peer0.org2.example.com:7051
        - CORE_PEER_CHAINCODELISTENADDRESS=peer0.org2.example.com:7052
        - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org2.example.com:7051
        - CORE_PEER_LOCALMSPID=Org2MSP

        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # the following setting starts chaincode containers on the same
        # bridge network as the peers
        # https://docs.docker.com/compose/networking/
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
```

```
        - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
        - CORE_PEER_GOSSIP_USELEADERELECTION=true
        - CORE_PEER_GOSSIP_ORGLEADER=false
        - CORE_PEER_PROFILE_ENABLED=false
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
      volumes:
          - /var/run/:/host/var/run/
          - ./crypto-config/peerOrganizations/org2.example.com/peers/peer0.org2.
example.com/msp:/etc/hyperledger/fabric/msp
          - ./crypto-config/peerOrganizations/org2.example.com/peers/peer0.org2.
example.com/tls:/etc/hyperledger/fabric/tls
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      command: peer node start
      ports:
        - 7051:7051
        - 7052:7052
        - 7053:7053
      depends_on:
        - couchdb
      networks:
        default:
          aliases:
            - aberic
      extra_hosts:
       - "orderer.example.com:172.31.159.130"

    cli:
      container_name: cli
      image: hyperledger/fabric-tools
      tty: true
      environment:
        - GOPATH=/opt/gopath
        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_PEER_ID=cli
        - CORE_PEER_ADDRESS=peer0.org2.example.com:7051
        - CORE_PEER_LOCALMSPID=Org2MSP
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/server
```

```
.crt
        - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
        - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      volumes:
          - /var/run/:/host/var/run/
          - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/aberic/
chaincode/go
          - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
          - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/
channel-artifacts
      depends_on:
        - peer0.org2.example.com
      extra_hosts:
       - "orderer.example.com:172.31.159.130"
       - "peer0.org2.example.com:172.31.159.129"
```

注意：CORE_PEER_LOCALMSPID 的值在这里应该为 Org2MSP，这也与第 4.1 节中的 configtx.yaml 和 crypto-config.yaml 配置文件里的组织参数相对应。

接下来对频道以及智能合约的操作参照第 5.4 节即可，但这里在第 5.3 节安装智能合约时做了背书限制，具体可以参考第 4.5 节中的讲解。所以，如果完全按照第 5.4 节中的测试方案执行，会发现 invoke 操作无效，这一点在后续智能合约概述的章节中会有所分析，这里不再赘述。

最后一个 peer1.org2 节点的部署也不再单列章节，直接参考下面的 docker-peer12.yaml 配置文件源码：

```
  version: '2'

  services:

    couchdb:
      container_name: couchdb
      image: hyperledger/fabric-couchdb
      # Comment/Uncomment the port mapping if you want to hide/expose the CouchDB
service,
      # for example map it to utilize Fauxton User Interface in dev environments.
      ports:
```

```
        - "5984:5984"

    ca:
      container_name: ca
      image: hyperledger/fabric-ca
      environment:
        - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
        - FABRIC_CA_SERVER_CA_NAME=ca
        - FABRIC_CA_SERVER_TLS_ENABLED=false
        - FABRIC_CA_SERVER_TLS_CERTFILE=/etc/hyperledger/fabric-ca-server-config/ca.
org2.example.com-cert.pem
        - FABRIC_CA_SERVER_TLS_KEYFILE=/etc/hyperledger/fabric-ca-server-
config/638c62f6073cf84d4a897b90fa31c0d53c4c2ffaa0b0be1c3c7df233eb33a27a_sk
      ports:
        - "7054:7054"
      command: sh -c 'fabric-ca-server start --ca.certfile /etc/hyperledger/fabric-
ca-server-config/ca.org2.example.com-cert.pem --ca.keyfile /etc/hyperledger/fabric-ca-
server-config/638c62f6073cf84d4a897b90fa31c0d53c4c2ffaa0b0be1c3c7df233eb33a27a_sk -b
admin:adminpw -d'
      volumes:
        - ./crypto-
config/peerOrganizations/org2.example.com/ca/:/etc/hyperledger/fabric-ca-server-config

    peer1.org2.example.com:
      container_name: peer1.org2.example.com
      image: hyperledger/fabric-peer
      environment:
        - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
        - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb:5984

        - CORE_PEER_ID=peer1.org2.example.com
        - CORE_PEER_NETWORKID=aberic
        - CORE_PEER_ADDRESS=peer1.org2.example.com:7051
        - CORE_PEER_CHAINCODELISTENADDRESS=peer1.org2.example.com:7052
        - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer1.org2.example.com:7051
        - CORE_PEER_LOCALMSPID=Org2MSP

        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # the following setting starts chaincode containers on the same
        # bridge network as the peers
        # https://docs.docker.com/compose/networking/
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
```

```
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
        - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
        - CORE_PEER_GOSSIP_USELEADERELECTION=true
        - CORE_PEER_GOSSIP_ORGLEADER=false
        - CORE_PEER_PROFILE_ENABLED=false
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
      volumes:
          - /var/run/:/host/var/run/
          - ./crypto-config/peerOrganizations/org2.example.com/peers/peer1.org2.
example.com/msp:/etc/hyperledger/fabric/msp
          - ./crypto-config/peerOrganizations/org2.example.com/peers/peer1.org2.
example.com/tls:/etc/hyperledger/fabric/tls
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      command: peer node start
      ports:
        - 7051:7051
        - 7052:7052
        - 7053:7053
      depends_on:
        - couchdb
      networks:
        default:
          aliases:
            - aberic
      extra_hosts:
       - "orderer.example.com:172.31.159.130"

    cli:
      container_name: cli
      image: hyperledger/fabric-tools
      tty: true
      environment:
        - GOPATH=/opt/gopath
        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_PEER_ID=cli
        - CORE_PEER_ADDRESS=peer1.org2.example.com:7051
        - CORE_PEER_LOCALMSPID=Org2MSP
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
```

```
crypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt
        - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      volumes:
          - /var/run/:/host/var/run/
          - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/aberic/
chaincode/go
          - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
          - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/
channel-artifacts
      depends_on:
        - peer1.org2.example.com
      extra_hosts:
       - "orderer.example.com:172.31.159.130"
       - "peer1.org2.example.com:172.31.143.21"
```

第 5.3 至本节中的 Peer 节点的配置文件基本一样，主要区别在于 Peer 节点的 IP 以及 MSPID 的值不同，这些配置代码可以参考。但读者在编写自己的配置文件时，需要按照实际的服务器配置进行设置。

## 5.6　本章小结

本章通过纯手动的方式进行了一次 Fabric 网络多台服务器组网的最小单元开发测试。

通过本章，对于 HyperLedger Fabric 跨服务器组网操作会有更加深入的理解，同时其中有些容易出现的错误也有所提及。

但其中关于 YAML 配置文件编写规范及参数含义等与 HyperLedger Fabric 相关性较低的概念尚未提及，这些概念可以通过网络或图书等其他渠道得到很好的学习，且这也不是本书所关注的重点，在阅读 YAML 时需要有一定的基本思想。

---

**注意：**本章所有 YAML 文件源码及所生成的配置文件等均可在本书前言的“读者服务”中找到并进行下载。

---

# 第 6 章　Kafka 集群部署

根据前面章节的介绍，知道了 Fabric 组网过程的第一步是需要生成证书等文件，而这些默认配置信息的生成依赖于 configtx.yaml 及 crypto-config.yaml 配置文件。

同样，在采用 Kafka 作为启动类型的 Fabric 网络中，configtx.yaml 及 crypto-config.yaml 配置文件依然有着重要地位，且其中的配置样本与先前的内容都会有些不同。

本章是以 Kafka 集群部署为案例开始讲述，在此之前，请先阅读前面的章节。

在阅读本章的时候会涉及一些概念性问题，这些概念性问题从第 3 章开始就屡次提及，但第 3、4、5 章中的目的主要是跑通一个最小单元的 Fabric 网络，需要读者首先对 HyperLedger Fabric 的网络进行一次基本的理解，要知晓 HyperLedger Fabric 网络在实际生产过程中可能应用到哪些场景中。

而本章将开启基于 Kafka 集群的部署，其中重要的概念是对前三章的总结，也是对本章及后续章节关于智能合约及 CouchDB 的铺垫。

## 6.1　Fabric账本

### 1. 账本（Ledger）

即所有的状态变更（state transitions）是有序且不可篡改的。状态变更是由参与方提交的 chaincode（智能合约）调用事务（transactions）的结果。每个事务都将产生一组资产键-值对，这些键-值对用于创建、更新或删除而提交给账本。

账本由 BlockChain（区块链）（“chain”）组成，区块则用来存储有序且不可篡改的记录，以及保存当前状态的状态数据库（state database）。在每一个 Channel 中都会存在一个账本。每一个 Peer 都会维护它作为其中成员的每一个 Channel 中的本地复制的账本。

### 2. 链（chain）

链是一个事务日志，是一个由 Hash 链接各个区块的结构，其中每个区块都包含了 $N$ 个事务的序列。

区块 header 包含了该区块的事务的 Hash，以及上一个区块头的 Hash。这样，所有在账本上的交易都是按顺序排列的，并以密码方式链接在一起。换句话说，在不破坏 Hash 链接的情

况下篡改账本数据是不可能的。最近的区块 Hash 代表了以前的每个事务，从而确保所有的 Peers 都处于一致和可信的状态。

链存储在 Peer 文件系统（本地或附加存储）上，有效地支持 BlockChain 工作负载的应用程序的特性。

3. 状态数据库

该账本的当前状态数据表示链事务日志中包含的所有键的最新值。

由于当前状态表示 Channel 所知道的全部最新键值，因此有时也称为“World State（世界状态）”。

在 Chaincode 调用对当前状态数据执行操作的事务时，为了使这些 Chaincode 交互非常有效，所有键的最新值都存储在一个状态数据库中。状态数据库只是一个索引视图到链的事务日志，因此可以在任何时候从链中重新生成它。在事务被接受之前，状态数据库将自动恢复（或在需要时生成）。

状态数据库包括 LevelDB 和 CouchDB。LevelDB 是嵌入在 Peer 进程中的默认状态数据库，并将 Chaincode 数据存储为键-值对。CouchDB 是一个可选的外部状态数据库，当所写的 Chaincode 数据被建模为 JSON 时，它提供了额外的查询支持，允许对 JSON 内容进行丰富的查询。

4. 事务流（transaction flow）

在高层业务逻辑处理上，事务流是由应用程序客户端发送的事务协议，该协议最终发送到指定的背书节点。

背书节点会验证客户端的签名，并执行一个 Chaincode 函数来模拟事务。最终返回给客户端的是 Chaincode 结果，即一组在 Chaincode 读集中读取的键-值版本，以及在 Chaincode 写集中写入的键-值集合，返回该 Peer 执行 Chaincode 后模拟出来的读写集结果，同时还会附带一个背书签名。

客户端将背书组合成一个事务 payload，并将其广播至一个 ordering service（排序服务节点），ordering service 为当前 Channel 上的所有 Peers 提供排序服务并生成区块。

实际上，客户端在将事务广播到排序服务之前，先将本次请求提交到 Peer，由 Peer 验证事务。

首先，Peer 将检查背书策略，以确保指定的 Peer 的正确分配已经签署了结果，并且将根据事务 payload 对签名进行身份验证。

其次，Peer 将对事务读取集进行版本控制，以确保数据完整性，并防止诸如重复开销之类

的问题。Hyperledger Fabric 具有并发控制，即事务允许并行执行（通过背书）来增加吞吐量，并且在提交（所有 Peer）的情况下，每个事务都经过验证，以确保没有其他事务修改它已经读取的数据。换句话说，它确保了在执行（批准）时间之后读取的数据没有发生变化，因此执行结果仍然有效，并且可以提交到账本状态数据库。如果读取的数据被另一个事务更改，则该区块中的相同事务被标记为无效，并且不应用于账本状态数据库。客户端应用程序被警告，并且可以在适当的情况下处理错误或重试。

---

**注意：**上述最后一段话的逻辑理论上是正确的，即先读取本地版本，然后根据本地版本发送广播至排序服务，再由排序服务进行事务处理。但事务处理结果通过实际使用 SDK 开发，该结果并未即时返回给当前调用客户端，即客户端无法实时获取事务状态，只能通过再次查询来确认最终结果。后续版本 SDK 可能会修复此问题。

---

## 6.2 事务处理流程

本章会介绍在标准资产交换过程中发生的事务机制。这个场景包括两个客户，A 和 B，他们在购买和销售萝卜（产品）。他们每个人在网络上都有一个 Peer，通过这个网络，他们发送自己的交易，并与 Ledger（账本）进行交互，如图 6-1 所示。

图6-1 交易

假设这个事务流中有一个 Channel 被设置并运行。应用程序客户端及该组织的证书颁发机构（CA）均已注册，并获得了必要的加密材料，用于对网络进行身份验证。

Chaincode（包含一组表示萝卜市场的初始状态的键-值对）被安装在 Peers 上，并在 Channel 上实例化。Chaincode 包含定义一组事务指令的逻辑，以及一个萝卜商定的价格。该 Chaincode 也已确定了一个背书策略，即 peerA 和 peerB 都必须支持任何交易，如图 6-2 所示。

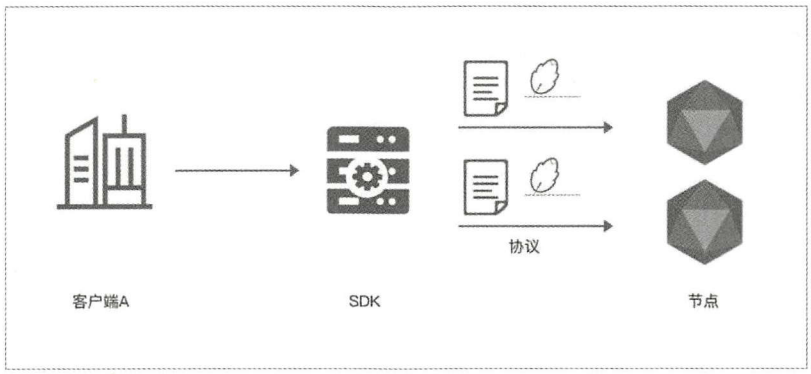


图6-2　安装交易过程

## 6.2.1　客户端发起事务

事务的发生过程——客户 A 正在发送一个请求，即购买萝卜。该请求的目标是 peerA 和 peerB，它们分别代表客户 A 和客户 B。背书策略规定，双方都必须认可任何交易，因此请求将被发送到 peerA 和 peerB。

接下来，将构造事务协议。使用任何一个被 HyperLedger Fabric 支持的 SDK（node、Java、Python）的应用程序创建一个可用的 API 来生成一个事务协议。协议是请求调用 Chaincode 函数，以便可以读取和（或）写入数据（例如，为资产编写新的键-值对）。SDK 作为一个 shim 来将事务协议打包成适当的格式（在 gRPC 上的协议缓冲区），并使用用户的加密凭证来为这个事务协议生成一个惟一的签名，如图 6-3 所示。

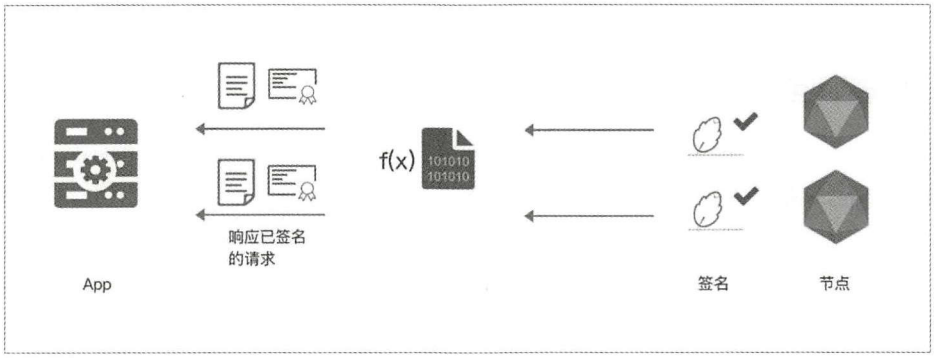


图6-3　事务回执

### 6.2.2 验证签名并执行事务

背书 peers 验证内容：

- 事务的协议是完整的。
- 在过去尚未被提交过（再现攻击保护）。
- 签名是有效的（使用 MSP）。
- 提交者（客户端，在这个例子中）是正确授权执行该操作在 Channel 中（也就是说，每个背书 peers 都确保提交者满足 Channel 的写入策略）。

MSP 是 peer 的一个组件，允许它们验证从客户端到达的事务请求，并签署事务结果（背书）。编写策略是在 Channel 创建时定义的，并确定哪个用户有权向该 Channel 提交事务，如图 6-4 所示。

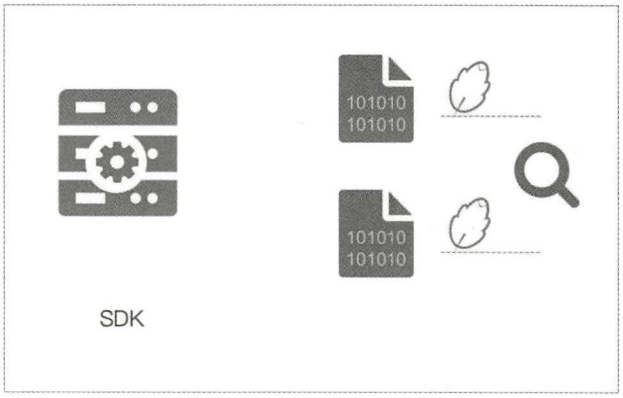


图6-4 节点背书

### 6.2.3 检查返回协议

应用程序验证背书 peer 的签名，并对提案响应进行比较，以确定提案的响应是否相同。如果 Chaincode 只是查询了账本，应用程序将检查查询响应结果，并且通常不会将查询事务提交给 orderer。如果客户端应用程序打算将事务提交到 orderer 来更新账本，则应用程序将确定在提交之前指定的背书策略是否已经完成（例如，peerA 和 peerB 都支持）。体系结构是这样的，即使应用程序选择不检查请求响应或转发未签署的事务，背书策略仍将由 peers 执行，并支持在提交阶段验证，如图 6-5 所示。

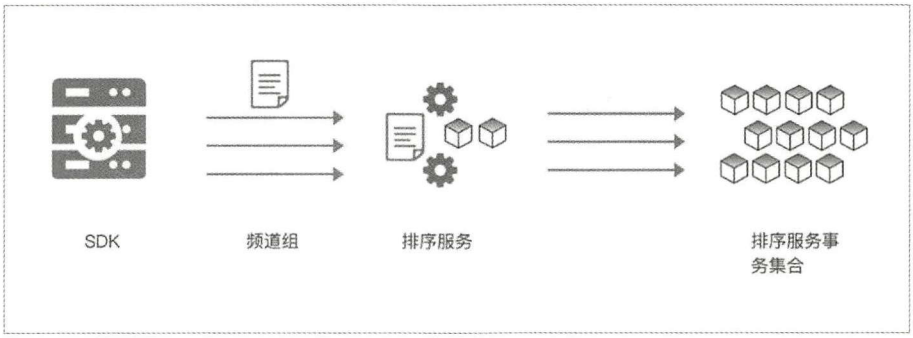


图6-5　协议流程

### 6.2.4　客户端将背书合并到交易中

应用程序将 transaction proposal（事务协议）和包含该“transaction message（事务消息）”的 peer 请求响应“广播”给 orderer 服务，该事务将包含 peer 请求返回的读写集、背书 peers 的签名以及 Channel ID。orderer 执行其操作无须检查该事务的全部内容，它只是从网络上的所有 Channels 中接收事务，对相同 Channel 中的事务按时间排序，并为每一个 Channel 中的一个或一列事务创建区块，如图 6-6 所示。

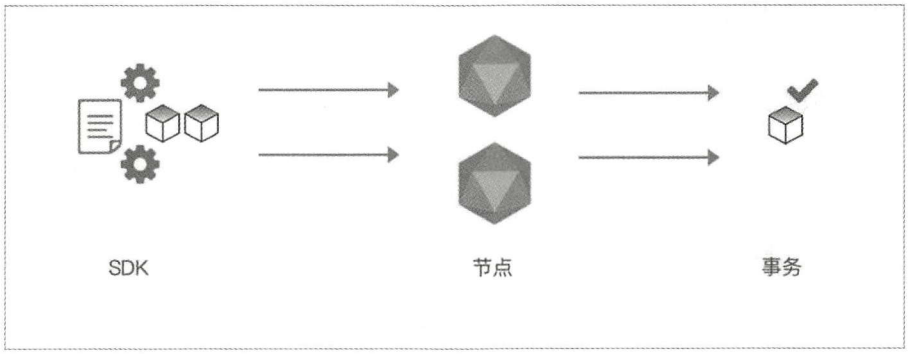


图6-6　交易处理

### 6.2.5　提交并验证事务

由事务集创建的区块将会被分发到 Channel 上所有的 Peers 中，在该区块中的事务集将被验证以确保满足背书策略，并确保在事务执行生成读取集之后，对读集变量的账本没有任何更改。区块中的事务集因此会被标记为有效或无效，如图 6-7 所示。

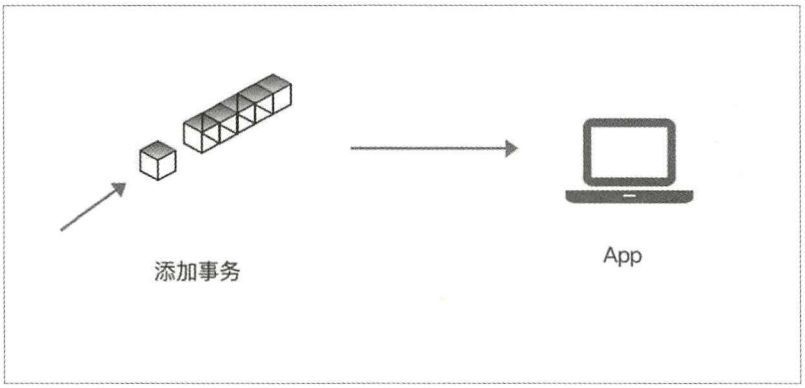


图6-7　事务验证

### 6.2.6　账本更新

每一个 Channel 都会将生成的区块追加到所属的链上，对于每个有效事务，都会将事务中的写集提交到当前状态数据库中。因前述而发出一个事务，通知客户端应用程序，事务（调用）已被追加到该链中，并通知该事务是否被验证或无效。

## 6.3　读写集规则

1. 事务模拟和读写集

客户端提交事务到 peer 节点，peer 节点会执行背书验证并模拟该事务的请求结果，为该事务的请求准备一个读写集。读集包含了该事务在读取本地账本时的一列事务版本信息及该信息对应的一列唯一键，写集包含了一个唯一键（可能也允许与读集中的键重复）列表和事务写入的最新值。如果事务的执行是删除操作，那么将为这一列唯一键设置一个 Delete 标记（在新值的位置）。

此外，如果事务为键多次写入一个值，则只保留最后一个写入值。另外，即使在事务读取结果发出之前更新了键值，也会返回事务读取已提交状态的值。换句话说，必须保证读写一致性。

正如上面提到的，键的版本只在读取集中记录，而写集仅包含由事务设置的一列唯一键和它们的最新值。

可以有各种各样的版本控制的方案。版本控制方案的最低要求是为给定的键生成不重复的标志符。例如，使用单调递增的版本可以采用一个基于区块链顶点的版本控制方案，在这个方案中，提交事务的顶点被用作事务修改后的所有键的最新版本，事务的顶点由一个元组表示（txNumber 是该块内事务的顶点）。该方案与增量编号方案相比，主要优点在于它支持其他组件，如 statedb、事务模拟和验证，以实现有效的设计选择。

下面是模拟假想事务的一个示例读写集的例子。为了简单起见，使用增量数字表示版本。

```
<TxReadWriteSet>
  <NsReadWriteSet name="chaincode1">
    <read-set>
      <read key="K1", version="1">
      <read key="K2", version="1">
    </read-set>
    <write-set>
      <write key="K1", value="V1">
      <write key="K3", value="V2">
      <write key="K4", isDelete="true">
    </write-set>
  </NsReadWriteSet>
<TxReadWriteSet>
```

此外，如果事务在模拟期间执行范围查询，范围查询及其结果将被添加到读写集作为查询信息。

2. 使用读写集进行事务验证和更新世界状态

提交者使用读写集的读集部分来检查事务的有效性，并使用读写集的写集部分更新受影响的键的版本和值。

在验证阶段，如果在事务读集中每一个 key 的版本都能够与 world state（假定之前所有的事务都是有效的，包括之前在同一区块中的已经被提交的事务）中的 key 版本一致，那么该条事务则被认为是有效的。如果读写集还包含一个或多个查询信息，则需要执行额外的验证。

这些额外的验证需要确保在本次事务查询信息返回结果的范围内的 key 没有被插入、删除或更新过，换句话说，如果在对提交状态的验证过程中重新执行任何一个范围查询（在模拟期间执行的事务），那么它应该会产生与在模拟时所观察到的结果相同的结果。此检查确保如果事务在提交期间观察到虚项，则该事务应被标记为无效。注意，这个虚项保护仅限于范围查询（例如：在 chaincode 中的 GetStateByRange 函数）并且还没有为其他查询（例如:在 chaincode 中的 GetQueryResult 函数）实现。其他查询有可能出现虚项，因此只能在没有提交到排序的只读事务中使用，除非应用程序能够保证模拟和验证或提交时间之间的结果集的稳定性。

如果一个事务通过了有效性检查，提交者将使用写集来更新世界状态。在更新阶段，对于写集中的每个键，相同键的值都设置为在写集中指定的值，进一步地，这个世界状态的键的版本会被改变，以反映最新的版本。

3. 模拟和验证案例

本节通过一个示例场景帮助理解语义。对于本例的目的，在世界状态中，键 k 的存在是由元组(k、ver、val)表示的，其中，ver 是键 k 的最新版本，它的值由 val 表示。

现在，考虑一组 5 个事务：T1、T2、T3、T4 和 T5，它们都在同一个快照上模拟世界状态。下面的代码片段显示了对事务进行模拟的世界状态的快照，以及由这些事务执行的读取和写入活动的顺序。

```
World state: (k1,1,v1), (k2,1,v2), (k3,1,v3), (k4,1,v4), (k5,1,v5)
T1 -> Write(k1, v1'), Write(k2, v2')
T2 -> Read(k1), Write(k3, v3')
T3 -> Write(k2, v2'')
T4 -> Write(k2, v2'''), read(k2)
T5 -> Write(k6, v6'), read(k5)
```

现在，假设这些事务是在 T1 的序列中排序的。T5 可以包含在一个区块或不同的区块中：

- T1 通过验证，因为它不执行任何读取操作。此外，世界状态中的键 k1 和 k2 的元组被更新为(k1，2，v1')和(k2，2，v2')。
- T2 失败了，因为它读取了一个键 k1，它被之前的事务修改为 T1。
- T3 通过验证，因为它不执行读操作。进一步的，在这个世界状态下的键的元组被更新到(k2，3，v2")。
- T4 失败了，因为它读取了一个键 k2，它被之前的事务 T1 修改过。
- T5 通过验证，因为它读取了一个键 k5，它没有被前面的任何事务修改过。

---

**注意**：目前还不支持具有多个读写集的事务。

---

## 6.4 Kafka集群配置

有了前面 5 个章节的部署经验，再加上本章前三节的概述理论，将会容易理解本章的重点 Kafka 集群部署。

在进行本次方案之前，首先需要对 Kafka 集群的拓扑有些简单的了解。搭建 Kafka 集群的最小单位组成如下：

- 三个 Zookeeper 节点集群。
- 四个 Kafka 节点集群。
- 三个 Orderer 排序服务节点集群。
- 其他 Peer 节点。

以上集群至少需要 10 个服务节点提供集群服务，其余节点用于背书验证、提交及数据同步。

Kafka 是一个分布式消息系统，由 LinkedIn 使用 scala 编写，用作 LinkedIn 的活动流（Activity Stream）和运营数据处理管道（Pipeline）的基础。具有高水平扩展和高吞吐量。

在 Fabric 网络中，数据是由 Peer 节点提交到 Orderer 排序服务，而 Orderer 相对于 Kafka 来说相当于上游模块，且 Orderer 还兼具提供了对数据进行排序及生成符合配置规范及要求的区块。而使用上游模块的数据计算、统计、分析，这个时候就可以使用类似于 Kafka 这样的分布式消息系统来协助业务流程。

有人说 Kafka 是一种共识模式，也就是说平等信任，所有的 HyperLedger Fabric 网络加盟方都是可信方，因为消息总是均匀地分布在各处。但具体生产使用的时候是依赖于背书来做到确权，相对而言，Kafka 应该只能是一种启动 Fabric 网络的模式或类型。

Zookeeper 是一种在分布式系统中被广泛用来作为分布式状态管理、分布式协调管理、分布式配置管理和分布式锁服务的集群。Kafka 增加和减少服务器都会在 Zookeeper 节点上触发相应的事件，Kafka 系统会捕获这些事件，进行新一轮的负载均衡，客户端也会捕获这些事件来进行新一轮的处理。

Orderer 排序服务是 Fabric 网络事务流中的最重要的环节，也是所有请求的终点，它并不会立刻对请求给予回馈，一是因为生成区块的条件所限，二是因为依托下游集群的消息处理需要等待结果。Kafka 集群拓扑图如图 6-8 所示。

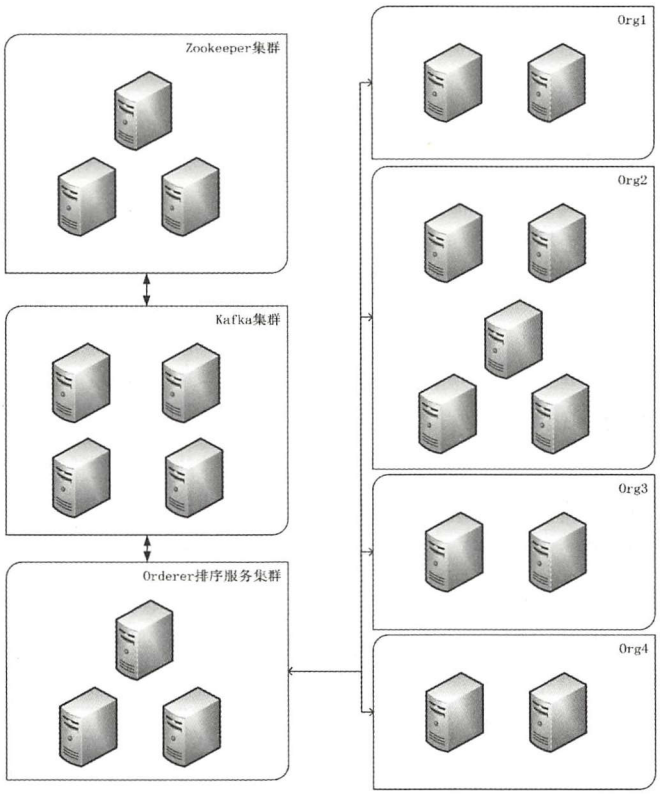


图6-8　Kafka集群拓扑图

本书根据实际需求，启用了如下 12 台服务器，见表 6-1。

表 6-1　集群配置表

| 名　称 | IP | Hostname | 组织机构 |
|---|---|---|---|
| Zk1 | 172.31.159.137 | zookeeper1 | |
| Zk2 | 172.31.159.135 | zookeeper2 | |
| Zk3 | 172.31.159.136 | zookeeper3 | |
| Kafka1 | 172.31.159.133 | kafka1 | |
| Kafka2 | 172.31.159.132 | kafka2 | |
| Kafka3 | 172.31.159.134 | kafka3 | |
| Kafka4 | 172.31.159.131 | kafka4 | |
| Orderer0 | 172.31.159.130 | orderer0.example.com | |
| Orderer1 | 172.31.143.22 | orderer1.example.com | |
| Orderer2 | 172.31.143.23 | orderer2.example.com | |
| peer0 | 172.31.159.129 | peer0.org1.example.com | Org1 |
| peerX | 172.31.143.21 | x.x.example.com | Org2 |

如同前述拓扑图及 Kafka 集群网络最小单元所述，所启用的服务器为最小配置单元。如果考虑到高可用性，可以自行学习参考 K8s 管理 Docker 的方案。

12 台服务器中 peer0 服务器是 peer0org1 节点服务器，而 peerX 则是任意可选节点服务器，用于测试各种节点的可能性。

在这些服务器当中，每一台都会安装 Docker、Docker-Compose 环境，而 Orderer 排序服务器及 Peer 节点服务器会额外地安装 Go 语言及 Fabric 环境。

当需要生成必要的配置文件时，任选其中一台部署有 Fabric 环境的服务器即可，本书选择 Orderer0 排序服务器作为配置文件生成服务器，即 172.31.159.130 服务器。

---

**注意**：*在没有特殊说明的情况下，所有的安装有 Fabric 环境的服务器部署方案均与第 5 章保持一致。*

---

### 6.4.1　crypto-config.yaml配置

前面章节中的 crypto-config.yaml 等没有被贴出来的配置文件都是采用 Fabric1.0 版本源码 e2e_cli 案例中的，从 Kafka 集群配置开始，将要准备编写属于自己的 crypto-config.yaml 配置文件。

还是先从源码开始，如下所示：

```
OrdererOrgs:
  - Name: Orderer
    Domain: example.com
    Specs:
      - Hostname: orderer0
      - Hostname: orderer1
      - Hostname: orderer2

PeerOrgs:
  - Name: Org1
    Domain: org1.example.com
    Template:
      Count: 2
    Users:
      Count: 1

  - Name: Org2
    Domain: org2.example.com
    Template:
      Count: 2
```

```
    Users:
      Count: 1
    Specs:
      - Hostname: foo
        CommonName: foo27.org2.example.com
      - Hostname: bar
      - Hostname: baz


  - Name: Org3
    Domain: org3.example.com
    Template:
      Count: 2
    Users:
      Count: 1

  - Name: Org4
    Domain: org4.example.com
    Template:
      Count: 2
    Users:
      Count: 1

  - Name: Org5
    Domain: org5.example.com
    Template:
      Count: 2
    Users:
      Count: 1
```

在上述配置源码中，Org1 即组织 1 的配置，与第 5 章的一样，Org3、Org4 和 Org5 是新增的组织配置信息，结构与 Org1 一致。与拓扑图相比，这里多出了一个 Org5，但普通 Peer 节点的多少并不影响集群部署的整体方案。

其中，Org2 加入了一些小的变化，采用了一种混合编排组织配置的方案，既有模板配置也有自定义配置。模板配置内容与 Org1、Org3、Org4 及 Org5 一样。自定义配置主要是新增了三个节点且名称分别是 foo、bar 和 baz，而 foo 又自定义了其生成的具体节点名称后缀，姑且理解为编号。

将编写完成的配置文件上传至 172.31.159.130 服务器/home/docker/github.com/hyperledger/fabric/aberic 目录下，并执行如下命令生成节点所需配置文件：

```
./bin/cryptogen generate --config=./crypto-config.yaml
```

根据第 4.1 节所述，该命令执行完成后会在/home/docker/github.com/hyperledger/fabric/aberic/crypto-config/peerOrganizations 目录下看到所有的组织配置文件，由于在编写 org2 的时候做了一些自定义，可以进入/home/docker/github.com/hyperledger/fabric/aberic/crypto-config/peerOrganizations/org2.example.com/peers 目录下查看自定义节点的目录信息，如图 6-9 所示。


图6-9　org2节点信息

在图中，除了默认模板生成的节点配置文件外，还有在 crypto-config.yaml 自定义的三个节点配置文件，说明自定义配置文件属性无误。且继续观察目录结构和内容可以发现，org1、org3、org4 和 org5 的结构是一致的，都是通过模板配置生成的。

## 6.4.2　configtx配置

通过第 4、5 两章的内容，知道了 configtx.yaml 配置文件在上下文中需要与 crypto-config.yaml 配置文件相匹配，同时该配置用于生成创世区块及设定 Fabric 网络启动类型。

由于本次采用的是 Kafka 集群部署，所以在本次文件配置中的启动类型应该为“kafka”。此外，还需要在 Addresses 中将 Orderer 可用排序服务即集群排序服务器的地址补全，在 Kafka 的 Brokers 中可填写非全量 Kafka 集群所用服务器 IP 或域名。

在组织配置中，需要将新增的三个组织配置编写入内，且所设置的锚节点必须是在生成范围中的某一节点。

具体的配置源码如下：

```
Profiles:

    TwoOrgsOrdererGenesis:
        Orderer:
            <<: *OrdererDefaults
            Organizations:
                - *OrdererOrg
        Consortiums:
            SampleConsortium:
```

```
                Organizations:
                    - *Org1
                    - *Org2
                    - *Org3
                    - *Org4
                    - *Org5
    TwoOrgsChannel:
        Consortium: SampleConsortium
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Org1
                - *Org2
                - *Org3
                - *Org4
                - *Org5

Organizations:

    - &OrdererOrg
        Name: OrdererOrg
        ID: OrdererMSP
        MSPDir: crypto-config/ordererOrganizations/example.com/msp

    - &Org1
        Name: Org1MSP
        ID: Org1MSP

        MSPDir: crypto-config/peerOrganizations/org1.example.com/msp

        AnchorPeers:
            - Host: peer0.org1.example.com
              Port: 7051

    - &Org2
        Name: Org2MSP
        ID: Org2MSP

        MSPDir: crypto-config/peerOrganizations/org2.example.com/msp

        AnchorPeers:
            - Host: peer0.org2.example.com
              Port: 7051
```

```
    - &Org3
        Name: Org3MSP
        ID: Org3MSP

        MSPDir: crypto-config/peerOrganizations/org3.example.com/msp

        AnchorPeers:
            - Host: peer0.org3.example.com
              Port: 7051

    - &Org4
        Name: Org4MSP
        ID: Org4MSP

        MSPDir: crypto-config/peerOrganizations/org4.example.com/msp

        AnchorPeers:
            - Host: peer0.org4.example.com
              Port: 7051

    - &Org5
        Name: Org5MSP
        ID: Org5MSP

        MSPDir: crypto-config/peerOrganizations/org5.example.com/msp

        AnchorPeers:
            - Host: peer0.org5.example.com
              Port: 7051

Orderer: &OrdererDefaults

    OrdererType: kafka

    Addresses:
        - orderer0.anti-moth.com:7050
        - orderer1.anti-moth.com:7050
        - orderer2.anti-moth.com:7050

    BatchTimeout: 2s

    BatchSize:

        MaxMessageCount: 10
```

```
        AbsoluteMaxBytes: 98 MB

        PreferredMaxBytes: 512 KB

    Kafka:
        Brokers:
            - 172.31.159.131:9092
            - 172.31.159.132:9092
            - 172.31.159.133:9092
            - 172.31.159.134:9092

    Organizations:

Application: &ApplicationDefaults

    Organizations:

Capabilities:
    Global: &ChannelCapabilities
        V1_1: true

    Orderer: &OrdererCapabilities
        V1_1: true

    Application: &ApplicationCapabilities
        V1_1: true
```

将编写完成的配置文件上传至 172.31.159.130 服务器/home/docker/github.com/ hyperledger/fabric/aberic 目录下，并执行如下命令生成创世区块文件：

```
./bin/configtxgen -profile TwoOrgsOrdererGenesis -outputBlock ./channel-artifacts/genesis.block
```

创世区块 genesis.block 是为了 Orderer 排序服务启动时用到的，Peer 节点在启动后需要创建的 Channel 的配置文件在这里也一并生成，执行具体命令和综合结果如下：

```
./bin/configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel-artifacts/mychannel.tx -channelID mychannel
```

当第 6.1.1 节和本节的配置文件生成完成后，以 Kafka 集群为基础的 Fabric 组网工程的第一步已经完成，接下来需要从顶层开始向下建设整个网络。

### 6.4.3　Zookeeper配置

根据表 6-1 中，Zookeeper 被分别命名为 Zk1、Zk2 和 Zk3，即需要三份配置文件以用来一一启动各 Zookeeper 节点服务。

这三份 Zookeeper 启动配置文件内容类似，首先看源码，docker-zookeeper1.yaml 配置文件如下所示：

```
version: '2'

services:

  zookeeper1:
    container_name: zookeeper1
    hostname: zookeeper1
    image: hyperledger/fabric-zookeeper
    restart: always
    environment:
      - ZOO_MY_ID=1
      - ZOO_SERVERS=server.1=zookeeper1:2888:3888 server.2=zookeeper2:2888:3888
server.3=zookeeper3:2888:3888
    ports:
      - "2181:2181"
      - "2888:2888"
      - "3888:3888"
    extra_hosts:
     - "zookeeper1:172.31.159.137"
     - "zookeeper2:172.31.159.135"
     - "zookeeper3:172.31.159.136"
     - "kafka1:172.31.159.133"
     - "kafka2:172.31.159.132"
     - "kafka3:172.31.159.134"
     - "kafka4:172.31.159.131"
```

要理解上述配置文件中的含义，先要对 ZooKeeper 的基本运转流程有所了解，主要有 5 个要素：

（1）选举 Leader。

（2）同步数据。

（3）选举 Leader 过程中算法有很多，但要达到的选举标准是一致的。

（4）Leader 要具有最高的执行 ID，类似 Root 权限。

（5）集群中大多数的机器得到响应并跟随选出的 Leader。

配置文件中的内容始终服从于上述 5 个要素。

其中，ZOO_MY_ID 参数是指当前所运行 Zookeeper 集群中的 zookeeper 当前服务节点的 ID，这个 ID 在集群中必须是唯一的并且必须有一个值，这个值只能在 1～255 之间择其一。

ZOO_SERVERS 是组成 Zookeeper 集群的服务器列表。客户端使用的列表必须与 ZooKeeper 服务器列表所拥有的每一个 Zookeeper 服务器相匹配。参数值列表中每一个服务的值都附带两个端口号，第一个是追随者用来连接 Leader 所使用的，第二个是用于选举 Leader。

因为在 ZOO_SERVERS 参数中的值里都使用的是服务器名，所以在 extra_hosts 中需要将所对应的服务器名所指向的具体 IP 和 Port 都显示出来，以提供给当前节点服务访问。

因此，docker-zookeeper2.yaml 及 docker-zookeeper3.yaml 的配置文件分别如下：

```
version: '2'

services:

  zookeeper2:
    container_name: zookeeper2
    hostname: zookeeper2
    image: hyperledger/fabric-zookeeper
    restart: always
    environment:
      - ZOO_MY_ID=2
      - ZOO_SERVERS=server.1=zookeeper1:2888:3888 server.2=zookeeper2:2888:3888
server.3=zookeeper3:2888:3888
    ports:
      - "2181:2181"
      - "2888:2888"
      - "3888:3888"
    extra_hosts:
     - "zookeeper1:172.31.159.137"
     - "zookeeper2:172.31.159.135"
     - "zookeeper3:172.31.159.136"
     - "kafka1:172.31.159.133"
     - "kafka2:172.31.159.132"
     - "kafka3:172.31.159.134"
     - "kafka4:172.31.159.131"
```

和

```
version: '2'

services:
```

```
    zookeeper3:
      container_name: zookeeper3
      hostname: zookeeper3
      image: hyperledger/fabric-zookeeper
      restart: always
      environment:
        - ZOO_MY_ID=3
        - ZOO_SERVERS=server.1=zookeeper1:2888:3888 server.2=zookeeper2:2888:3888
server.3=zookeeper3:2888:3888
      ports:
        - "2181:2181"
        - "2888:2888"
        - "3888:3888"
      extra_hosts:
       - "zookeeper1:172.31.159.137"
       - "zookeeper2:172.31.159.135"
       - "zookeeper3:172.31.159.136"
       - "kafka1:172.31.159.133"
       - "kafka2:172.31.159.132"
       - "kafka3:172.31.159.134"
       - "kafka4:172.31.159.131"
```

随后在启动中通过观察其日志，可以清晰地捕捉到 Zookeeper 枚举 Leader 的整个过程。

---

**注意**：Zookeeper 集群的数量可以是 3、5 或 7，它的值需要是一个奇数以避免 split-brain 的情况，同时选择大于 1 的值为了避免单点故障，如果集群数量超过 7 个 ZooKeeper 服务将会被认为 overkill，即无法承受。

---

### 6.4.4　Kafka配置

根据表 6-1，Kafka 被分别命名为 Kafka1、Kafka2、Kafka3 和 Kafka4，即需要四份配置文件启动各 Kafka 节点服务。

这四份 Kafka 启动配置文件内容类似，首先看源码，docker-kafka1.yaml 配置文件如下所示：

```
version: '2'

services:

  kafka1:
    container_name: kafka1
```

```
    hostname: kafka1
    image: hyperledger/fabric-kafka
    restart: always
    environment:
      - KAFKA_BROKER_ID=1
      - KAFKA_MIN_INSYNC_REPLICAS=2
      - KAFKA_DEFAULT_REPLICATION_FACTOR=3
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper1:2181,zookeeper2:2181,zookeeper3:2181
      - KAFKA_MESSAGE_MAX_BYTES=103809024
      - KAFKA_REPLICA_FETCH_MAX_BYTES=103809024
      - KAFKA_UNCLEAN_LEADER_ELECTION_ENABLE=false
      - KAFKA_LOG_RETENTION_MS=-1
    ports:
      - "9092:9092"
    extra_hosts:
     - "zookeeper1:172.31.159.137"
     - "zookeeper2:172.31.159.135"
     - "zookeeper3:172.31.159.136"
     - "kafka1:172.31.159.133"
     - "kafka2:172.31.159.132"
     - "kafka3:172.31.159.134"
     - "kafka4:172.31.159.131"
```

Kafka 默认的端口号为 9092。

（1）KAFKA_BROKER_ID 是一个唯一的非负整数 ID 进行标识，这个 ID 可以作为代理（Broker）的“名字”，并且它的存在使得代理无须混淆消费者就可以迁移到不同的 Host/Port 上。可以选择任意喜欢的数字作为 ID，只要 ID 是唯一的即可。

（2）KAFKA_DEFAULT_REPLICATION_FACTOR 参数要与 KAFKA_MIN_I NSYNC_REPLICAS 参数放到一起来相互参考。

（3）KAFKA_DEFAULT_REPLICATION_FACTOR 的字面含义是默认复制因子，它是一个小于 Kafka 集群数量的值，这里假设 Kafka 集群数量是 $K$，而默认复制因子的值是 $N$，且 $N<K$。这里的 $N$ 代表每个 Channel 都保存 $N$ 个副本的数据到 Kafka 的代理上，这些都是一个 Channel 的 ISR（同步副本）集合的候选，因为并非所有的代理在任何时候都是可用的。如果少于 $N$ 个代理，Channel 的创建是不能成功的。因此，如果设置 $N$ 的值为 $K$，一个代理失效后，那么区块链网络将不能再创建新的 Channel，即 Orderer 排序服务的 Crash 容错也就不存在了。

这个配置的字面含义是最小同步备份，这里也假设它是一个为 $M$ 的值，它表示提交数据时会写入至少 $M$ 个副本（这些数据然后会被同步并且归属到同步副本集合或 ISR）。其他情况，写入操作会返回一个错误。如果 Channel 写入的数据多达 $N–M$ 个副本变得不可用，操作

可以正常执行。如果有更多的副本不可用，Kafka 不可以维护一个有 *M* 数量的 ISR 集合，因此 Kafka 停止接收写操作。Channel 只有当同步 *M* 个副本后才可以重新写。因此，*M* 是一个大于 1 且小于 *N* 的值。

（4）KAFKA_ZOOKEEPER_CONNECT 是指向 Zookeeper 节点的集合，其中包含 ZK 的集合。

（5）KAFKA_MESSAGE_MAX_BYTES 参数要与 KAFKA_REPLICA_FETCH_MAX_BYTES 参数放到一起来相互参考。

KAFKA_MESSAGE_MAX_BYTES 的字面含义是消息最大字节数，它与 configtx.yaml 的 AbsoluteMaxBytes 相对应。在 configtx.yaml 设置了每个区块最大有 Orderer.AbsoluteMaxBytes 个字节（不包括 Header），假定这里设置的值为 A（目前 98MB），那么 KAFKA_MESSAGE_MAX_BYTES 应该设置一个大于 A 的值，同时为 header 增加一些缓冲区空间——1MB 已经足够大。KAFKA_MESSAGE_MAX_BYTES、KAFKA_REPLICA_FETCH_MAX_BYTES 与 Orderer.AbsoluteMaxBytes 因满足如下关系：

```
AbsoluteMaxBytes < KAFKA_REPLICA_FETCH_MAX_BYTES <= KAFKA_MESSAGE_MAX_BYTES
```

更完整的是，KAFKA_MESSAGE_MAX_BYTES 应该严格小于 KAFKA_SOCKET_REQUEST_MAX_BYTES 的值，KAFKA_SOCKET_REQUEST_MAX_BYTES 在该配置中没有指定，其值默认被设置为 100MB。如果想要区块大于 100MB，需要编辑 fabric/orderer/kafka/config.go 文件里硬编码的值 brokerConfig.Producer.MaxMessageBytes，修改后重新编译源码得到二进制文件，这种设置官方是不建议的。

（6）KAFKA_REPLICA_FETCH_MAX_BYTES 字面含义是副本获取最大字节数，它是试图为每个 Channel 获取的消息的字节数。这不是绝对最大值，如果获取的信息大于这个值，则仍然会返回信息，以确保可以取得进展。代理所接受的最大消息大小是通过 KAFKA_MESSAGE_MAX_BYTES 定义的。

（7）KAFKA_UNCLEAN_LEADER_ELECTION_ENABLE 的字面含义是非一致性的 Leader 选举。数据一致性在区块链环境中是至关重要的。不能从 ISR 集合之外选取 Channel 的 Leader，否则将会面临对于之前的 Leader 产生的 offsets 覆盖的风险，这样的结果是，Orderers 排序服务产生的区块可能会重新写入区块链。

（8）KAFKA_LOG_RETENTION_MS 是对于压缩日志保留的最长时间，也是客户端消费消息的最长时间，它将控制压缩后的数据。除非 Orderer 排序服务对 Kafka 日志的修剪增加支持，否则需要关闭基于时间的日志保留方式并且避免分段到期。基于大小的日志保留方式 KAFKA_LOG_RETENTION_BYTES 在 Kafka 中已经默认关闭，因此不需要再次明确设置。

如上所述，docker-kafka2.yaml、docker-kafka3.yaml 及 docker-kafka4.yaml 的配置文件分别

如下：

docker-kafka2.yaml：

```
version: '2'

services:

  kafka2:
    container_name: kafka2
    hostname: kafka2
    image: hyperledger/fabric-kafka
    restart: always
    environment:
      - KAFKA_BROKER_ID=2
      - KAFKA_MIN_INSYNC_REPLICAS=2
      - KAFKA_DEFAULT_REPLICATION_FACTOR=3
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper1:2181,zookeeper2:2181,zookeeper3:2181
      - KAFKA_MESSAGE_MAX_BYTES=103809024
      - KAFKA_REPLICA_FETCH_MAX_BYTES=103809024
      - KAFKA_UNCLEAN_LEADER_ELECTION_ENABLE=false
      - KAFKA_LOG_RETENTION_MS=-1
    ports:
      - "9092:9092"
    extra_hosts:
     - "zookeeper1:172.31.159.137"
     - "zookeeper2:172.31.159.135"
     - "zookeeper3:172.31.159.136"
     - "kafka1:172.31.159.133"
     - "kafka2:172.31.159.132"
     - "kafka3:172.31.159.134"
     - "kafka4:172.31.159.131"
```

docker-kafka3.yaml：

```
version: '2'

services:

  kafka3:
    container_name: kafka3
    hostname: kafka3
    image: hyperledger/fabric-kafka
    restart: always
    environment:
      - KAFKA_BROKER_ID=3
```

```
      - KAFKA_MIN_INSYNC_REPLICAS=2
      - KAFKA_DEFAULT_REPLICATION_FACTOR=3
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper1:2181,zookeeper2:2181,zookeeper3:2181
      - KAFKA_MESSAGE_MAX_BYTES=103809024
      - KAFKA_REPLICA_FETCH_MAX_BYTES=103809024
      - KAFKA_UNCLEAN_LEADER_ELECTION_ENABLE=false
      - KAFKA_LOG_RETENTION_MS=-1
    ports:
      - "9092:9092"
    extra_hosts:
     - "zookeeper1:172.31.159.137"
     - "zookeeper2:172.31.159.135"
     - "zookeeper3:172.31.159.136"
     - "kafka1:172.31.159.133"
     - "kafka2:172.31.159.132"
     - "kafka3:172.31.159.134"
     - "kafka4:172.31.159.131"
```

docker-kafka4.yaml：

```
version: '2'

services:

  kafka4:
    container_name: kafka4
    hostname: kafka4
    image: hyperledger/fabric-kafka
    restart: always
    environment:
      - KAFKA_BROKER_ID=4
      - KAFKA_MIN_INSYNC_REPLICAS=2
      - KAFKA_DEFAULT_REPLICATION_FACTOR=3
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper1:2181,zookeeper2:2181,zookeeper3:2181
      - KAFKA_MESSAGE_MAX_BYTES=103809024
      - KAFKA_REPLICA_FETCH_MAX_BYTES=103809024
      - KAFKA_UNCLEAN_LEADER_ELECTION_ENABLE=false
      - KAFKA_LOG_RETENTION_MS=-1
    ports:
      - "9092:9092"
    extra_hosts:
     - "zookeeper1:172.31.159.137"
     - "zookeeper2:172.31.159.135"
     - "zookeeper3:172.31.159.136"
     - "kafka1:172.31.159.133"
```

```
      - "kafka2:172.31.159.132"
      - "kafka3:172.31.159.134"
      - "kafka4:172.31.159.131"
```

注意：Kafka 的最小值应该被设置为 4，这是为了满足 Crash 容错的最小节点数。如果有 4 个代理，则可以容错一个代理崩溃，即一个代理停止服务后，Channel 仍然可以继续读写，新的 Channel 可以被创建。

### 6.4.5 Orderer配置

根据表 6-1，Orderer 被分别命名为 Orderer0、Orderer1 和 Orderer2，即需要三份配置文件启动各 Orderer 节点服务。

这三份 Orderer 启动配置文件内容类似，首先看源码，docker-orderer0.yaml 配置文件如下所示：

```
version: '2'

services:

  orderer0.example.com:
    container_name: orderer0.example.com
    image: hyperledger/fabric-orderer
    environment:
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
      - ORDERER_GENERAL_LOGLEVEL=debug
      # - ORDERER_GENERAL_LOGLEVEL=error
      - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
      - ORDERER_GENERAL_LISTENPORT=7050
      #- ORDERER_GENERAL_GENESISPROFILE=AntiMothOrdererGenesis
      - ORDERER_GENERAL_GENESISMETHOD=file
      - ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block
      - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
      - ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
```



```
        #- ORDERER_GENERAL_LEDGERTYPE=ram
        #- ORDERER_GENERAL_LEDGERTYPE=file
        # enabled TLS
        - ORDERER_GENERAL_TLS_ENABLED=false
        - ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
        - ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
        - ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]


        - ORDERER_KAFKA_RETRY_LONGINTERVAL=10s
        - ORDERER_KAFKA_RETRY_LONGTOTAL=100s
        - ORDERER_KAFKA_RETRY_SHORTINTERVAL=1s
        - ORDERER_KAFKA_RETRY_SHORTTOTAL=30s
        - ORDERER_KAFKA_VERBOSE=true
        - ORDERER_KAFKA_BROKERS=[172.31.159.131:9092,172.31.159.132:9092,172.31.159.
133:9092,172.31.159.134:9092]
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric
      command: orderer
      volumes:
      - ../config/channel-artifacts/genesis.block:/var/hyperledger/orderer /orderer.
genesis.block
      - ../config/crypto-config/ordererOrganizations/example.com/orderers/orderer0.
example.com/msp:/var/hyperledger/orderer/msp
      - ../config/crypto-config/ordererOrganizations/example.com/orderers/orderer0.
example.com/tls/:/var/hyperledger/orderer/tls
      networks:
        default:
          aliases:
            - example
      ports:
        - 7050:7050
      extra_hosts:
       - "kafka1:172.31.159.133"
       - "kafka2:172.31.159.132"
       - "kafka3:172.31.159.134"
       - "kafka4:172.31.159.131"
```

1. 普通环境变量需要注意的配置

CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE 用来创建 Docker 容器的参数。容器可以使用 ipam 和 dns-server 来高效地创建集群网络模式——为容器设置网络模式。支持的标准值是："host"（默认）、"bridge"、"ipvlan"和"none"。

---

注意：CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE 如果按照配置文档介绍的标准值赋值可能会存在问题，比如引发智能合约实例化超时等错误。这里给的值是 aberic_default，属半自定义值，当 Orderer 排序服务容器启动的时候，其 network 会根据当前 YAML 所在目录的名称再加上_default 后缀生成默认的 network，因为当前 YAML 文件所在目录名为 aberic，所以是 aberic_default。

---

ORDERER_GENERAL_LOGLEVEL 设置当前程序的日志级别，当前设为 debug 是为了方便调试，生产中应该设置为 error 等较高级别，仅用来监听错误日志即可。

ORDERER_GENERAL_GENESISMETHOD 用于告知关于本 Fabric 网络的创世区块被包含在一个文件信息中。

ORDERER_GENERAL_GENESISFILE 与 ORDERER_GENERAL_GENESISMETHOD 对应，指定该创世区块的确切路径。

ORDERER_GENERAL_LOCALMSPID 是在 crypto-config.yaml 配置文件中定义在 Fabric 网络中的 MSP（会员服务提供者）的 ID，其值与 crypto-config.yaml 配置文件中的定义保持一致。

ORDERER_GENERAL_LOCALMSPDIR 与 ORDERER_GENERAL_LOCALMSPID 对应，指该 MSPID 目录的路径所在。

ORDERER_GENERAL_TLS_ENABLED 表示是否启用 TLS 以便以更加安全的传输方式在 Fabric 网络中进行通信。

ORDERER_GENERAL_TLS_PRIVATEKEY 是为服务器证书设置私钥文件的位置。

ORDERER_GENERAL_TLS_CERTIFICATE 是为 TLS 设置服务器证书的位置。

ORDERER_GENERAL_TLS_ROOTCAS 是为 TLS 设置根证书的位置。

ORDERER_KAFKA 系列的配置是作为一个 Kafka 生产者和消费者所需关注的配置，其中的 ORDERER_KAFKA_RETRY 系列配置允许调整 metadata/producer/consumer 请求的频率，同样可以设置超时时间。

ORDERER_KAFKA_RETRY_LONGINTERVAL 表示每间隔多长时间进行一次重试。

ORDERER_KAFKA_RETRY_LONGTOTAL 表示总共的重试时长是多少。



ORDERER_KAFKA_RETRY_SHORTINTERVAL 表示每间隔多长时间进行一次重试。

ORDERER_KAFKA_RETRY_SHORTTOTAL 表示总共的重试时长是多少。

其重试间隔时间和总时间约束是先短后长，即当执行创建新的 Channel 或加载一个已经存在的 Channel 时，Orderer 为该 Channel 对应的 Kafka 分区创建一个 producer（writer），使用 producer 发一个空操作连接信息到这个分区并为这个分区创建一个 consumer。

如果上述过程中的任意一个步骤失败了，则会发起重试，重试首先进行短频率流程，即每隔 ORDERER_KAFKA_RETRY_SHORTINTERVAL 时间发起一次重试，总计重试 ORDERER_KAFKA_RETRY_SHORTTOTAL 时长。如果还没有成功，则每隔 ORDERER_KAFKA_RETRY_LONGINTERVAL 时间发起一次重试，总计重试 ORDERER_KAFKA_RETRY_LONGTOTAL 时长。

---

**注意：**在上述步骤成功执行之前，Orderer 是不能对一个 Channel 进行读写操作的。

---

ORDERER_KAFKA_VERBOSE 表示启用日志与 Kafka 进行交互。

ORDERER_KAFKA_BROKERS 是指向 Kafka 节点的集合，其中包含 Kafka 的集合。

2. 工作路径配置

working_dir 用于设置 Orderer 排序服务的工作路径。

3. 卷宗

Volumes 表示为了映射在环境配置中使用的目录，这说明了 MSP、TLS 的键、ROOT 和 CERT 文件的位置，其中包含了在其中编码的创世纪块信息。

那么如上所述，docker-orderer1.yaml 及 docker-orderer2.yaml 的配置文件分别如下：

docker-orderer1.yaml：

```
version: '2'

services:

  orderer1.example.com:
    container_name: orderer1.example.com
    image: hyperledger/fabric-orderer
    environment:
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
      - ORDERER_GENERAL_LOGLEVEL=debug
      # - ORDERER_GENERAL_LOGLEVEL=error
      - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
```

```
        - ORDERER_GENERAL_LISTENPORT=7050
        #- ORDERER_GENERAL_GENESISPROFILE=AntiMothOrdererGenesis
        - ORDERER_GENERAL_GENESISMETHOD=file
        - ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block
        - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
        - ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
        #- ORDERER_GENERAL_LEDGERTYPE=ram
        #- ORDERER_GENERAL_LEDGERTYPE=file
        # enabled TLS
        - ORDERER_GENERAL_TLS_ENABLED=false
        - ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
        - ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
        - ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]

        - ORDERER_KAFKA_RETRY_LONGINTERVAL=10s
        - ORDERER_KAFKA_RETRY_LONGTOTAL=100s
        - ORDERER_KAFKA_RETRY_SHORTINTERVAL=1s
        - ORDERER_KAFKA_RETRY_SHORTTOTAL=30s
        - ORDERER_KAFKA_VERBOSE=true
        - ORDERER_KAFKA_BROKERS=[172.31.159.131:9092,172.31.159.132:9092,172.31.159.
133:9092,172.31.159.134:9092]
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric
      command: orderer
      volumes:
      - ../config/channel-artifacts/genesis.block:/var/hyperledger/orderer/orderer.
genesis.block
      - ../config/crypto-config/ordererOrganizations/example.com/orderers/orderer1.
example.com/msp:/var/hyperledger/orderer/msp
      - ../config/crypto-config/ordererOrganizations/example.com/orderers/orderer1.
example.com/tls/:/var/hyperledger/orderer/tls
      networks:
        default:
          aliases:
            - example
      ports:
        - 7050:7050
      extra_hosts:
       - "kafka1:172.31.159.133"
       - "kafka2:172.31.159.132"
       - "kafka3:172.31.159.134"
       - "kafka4:172.31.159.131"
```

docker-orderer2.yaml：

```
version: '2'
```



```
  services:

    orderer2.example.com:
      container_name: orderer2.example.com
      image: hyperledger/fabric-orderer
      environment:
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
        - ORDERER_GENERAL_LOGLEVEL=debug
        # - ORDERER_GENERAL_LOGLEVEL=error
        - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
        - ORDERER_GENERAL_LISTENPORT=7050
        #- ORDERER_GENERAL_GENESISPROFILE=AntiMothOrdererGenesis
        - ORDERER_GENERAL_GENESISMETHOD=file
        - ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer.genesis.block
        - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
        - ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
        #- ORDERER_GENERAL_LEDGERTYPE=ram
        #- ORDERER_GENERAL_LEDGERTYPE=file
        # enabled TLS
        - ORDERER_GENERAL_TLS_ENABLED=false
        - ORDERER_GENERAL_TLS_PRIVATEKEY=/var/hyperledger/orderer/tls/server.key
        - ORDERER_GENERAL_TLS_CERTIFICATE=/var/hyperledger/orderer/tls/server.crt
        - ORDERER_GENERAL_TLS_ROOTCAS=[/var/hyperledger/orderer/tls/ca.crt]

        - ORDERER_KAFKA_RETRY_LONGINTERVAL=10s
        - ORDERER_KAFKA_RETRY_LONGTOTAL=100s
        - ORDERER_KAFKA_RETRY_SHORTINTERVAL=1s
        - ORDERER_KAFKA_RETRY_SHORTTOTAL=30s
        - ORDERER_KAFKA_VERBOSE=true
        - ORDERER_KAFKA_BROKERS=[172.31.159.131:9092,172.31.159.132:9092,172.31.
159.133:9092,172.31.159.134:9092]
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric
      command: orderer
      volumes:
      - ../config/channel-artifacts/genesis.block:/var/hyperledger/orderer/orderer.
genesis.block
      - ../config/crypto-config/ordererOrganizations/example.com/orderers/orderer2.
example.com/msp:/var/hyperledger/orderer/msp
      - ../config/crypto-config/ordererOrganizations/example.com/orderers/orderer2.
example.com/tls/:/var/hyperledger/orderer/tls
      networks:
        default:
          aliases:
```

```
      - example
    ports:
      - 7050:7050
    extra_hosts:
     - "kafka1:172.31.159.133"
     - "kafka2:172.31.159.132"
     - "kafka3:172.31.159.134"
     - "kafka4:172.31.159.131"
```

## 6.5 启动集群

Kafka 集群的启动顺序由上至下，即根集群必须优先启动：先启动 Zookeeper 集群，随后是 Kafka 集群，最后是 Orderer 排序服务集群。

当所有集群都顺利启动后，可以开始向集群中新增业务节点并开启各项正式业务。

### 6.5.1 启动Zookeeper集群

在第 6.4 节中提到 Zookeeper 所在节点服务器只需要安装 Docker 及 Docker-Compose 环境，Go 语言及 Fabric 环境无需部署。但为了便于项目更加清晰，本书对此步骤操作依旧会在 172.31.159.137、172.31.159.135 和 172.31.159.136 三台服务器上建立一个/home/docker/github.com/hyperledger/fabric/aberic 目录。

分别将 docker-zookeeper1.yaml、docker-zookeeper2.yaml 和 docker-zookeeper3.yaml 上传至 172.31.159.137 、172.31.159.135 及 172.31.159.136 三台服务器的 /home/docker/github.com/hyperledger/fabric/aberic 目录下，如图 6-10 所示。

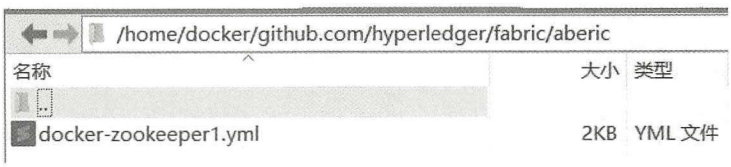


图6-10 Zookeeper目录

随后，在 172.31.159.137 服务器上执行如下命令启动 docker-zookeeper1.yaml：

```
docker-compose -f docker-zookeeper1.yml up
```

该命令后面没有加入“-d”参数，是为了直接查看 Zookeeper 的启动日志。

此时会有如下日志：

```
zookeeper1    | java.net.ConnectException: Connection refused (Connection refused)
zookeeper1    |     at java.net.PlainSocketImpl.socketConnect(Native Method)
```



```
    zookeeper1    |    at java.net.AbstractPlainSocketImpl.doConnect
(AbstractPlainSocketImpl.java:350)
    zookeeper1    |    at java.net.AbstractPlainSocketImpl.connectToAddress
(AbstractPlainSocketImpl.java:206)
    zookeeper1    |    at java.net.AbstractPlainSocketImpl.connect
(AbstractPlainSocketImpl.java:188)
    zookeeper1    |    at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)
    zookeeper1    |    at java.net.Socket.connect(Socket.java:589)
    zookeeper1    |    at org.apache.zookeeper.server.quorum.QuorumCnxManager.
connectOne(QuorumCnxManager.java:381)
    zookeeper1    |    at org.apache.zookeeper.server.quorum.QuorumCnxManager.
connectAll(QuorumCnxManager.java:426)
    zookeeper1    |    at org.apache.zookeeper.server.quorum.FastLeaderElection.
lookForLeader(FastLeaderElection.java:843)
    zookeeper1    |   at org.apache.zookeeper.server.quorum.QuorumPeer.Run (QuorumPeer.
java:822)
    zookeeper1    | 2018-04-16 05:18:23,680 [myid:1] - INFO  [QuorumPeer[myid=1] /0.0.
0.0:2181:QuorumPeer$QuorumServer@149] - Resolved hostname: zookeeper3 to address:
zookeeper3/172.31.159.136
    zookeeper1    | 2018-04-16 05:18:23,680 [myid:1] - INFO  [QuorumPeer[myid=1]/
0.0.0.0:2181:FastLeaderElection@852] - Notification time out: 51200
```

即无法注册或找到其他 Zookeeper 节点，这时再按照上述方案和命令依次启动 172.31.159.135 及 172.31.159.136 上的 Zookeeper。

再回头看 Zookeeper1 的日志如下：

```
    zookeeper1    | 2018-04-16 05:20:16,861 [myid:1] - INFO  [QuorumPeer[myid=1]/0.0.
0.0:2181:Follower@61] - FOLLOWING - LEADER ELECTION TOOK - 164213
    zookeeper1    | 2018-04-16 05:20:16,863 [myid:1] - INFO  [QuorumPeer[myid=1]/0.0.0.
0:2181:QuorumPeer$QuorumServer@149] - Resolved hostname: zookeeper2 to address:
zookeeper2/172.31.159.135
    zookeeper1    | 2018-04-16 05:20:16,891 [myid:1] - INFO  [QuorumPeer[myid=1]/0.
0.0.0:2181:Learner@326] - Getting a diff from the leader 0x0
    zookeeper1    | 2018-04-16 05:20:16,893 [myid:1] - INFO  [QuorumPeer[myid=1]
/0.0.0.0:2181:FileTxnSnapLog@240] - Snapshotting: 0x0 to /data/version-2/snapshot.0
    zookeeper1    | 2018-04-16 05:20:33,598 [myid:1] - INFO  [zookeeper1/172.18.
0.2:3888:QuorumCnxManager$Listener@541] - Received connection request
/172.31.159.136:38966
```

Zookeeper1 已经与 Zookeeper2 和 Zookeeper3 进行通信，并设定自身属性为 FOLLOWING。Zookeeper3 与 Zookeeper1 的类似，都会 FOLLOWING。

而 Zookeeper2 的日志如下：

```
    zookeeper2    | 2018-04-16 05:20:16,838 [myid:2] - INFO  [QuorumPeer[myid=2]
```

```
/0.0.0.0:2181:QuorumPeer@856] - LEADING
    zookeeper2    | 2018-04-16 05:20:16,842 [myid:2] - INFO  [QuorumPeer[myid=2]/
0.0.0.0:2181:Leader@59] - TCP NoDelay set to: true
    zookeeper2    | 2018-04-16 05:20:16,852 [myid:2] - INFO  [QuorumPeer[myid=2]/
0.0.0.0:2181:Environment@100] - Server environment:zookeeper.version=3.4.9-1757313,
built on 08/23/2016 06:50 GMT
    zookeeper2    | 2018-04-16 05:20:16,853 [myid:2] - INFO  [QuorumPeer[myid=2]/0.
0.0.0:2181:Environment@100] - Server environment:host.name=zookeeper2
    zookeeper2    | 2018-04-16 05:20:16,853 [myid:2] - INFO  [QuorumPeer[myid=2]/0.0.0.
0:2181:Environment@100] - Server environment:java.version=1.8.0_151
    zookeeper2    | 2018-04-16 05:20:16,853 [myid:2] - INFO  [QuorumPeer[myid=2]/0.0.0.
0:2181:Environment@100] - Server environment:java.vendor=Oracle Corporation
    zookeeper2    | 2018-04-16 05:20:16,853 [myid:2] - INFO  [QuorumPeer[myid=
2]/0.0.0.0:2181:Environment@100] - Server environment:java.home=/usr/lib/jvm/java-8-
openjdk-amd64/jre
```

通过选举，Zookeeper2 成了新的 Leader。

现在进入 Zookeeper2 所在 172.31.159.135 服务器，执行如下命令进入 Zookeeper 容器内部：

```
docker exec -it zookeeper2 bash
```

随后进入 bin 目录，并执行如下命令查看当前 Zookeeper 的状态：

```
zkServer.sh status
```

可以得到如下日志信息：

```
ZooKeeper JMX enabled by default
Using config: /conf/zoo.cfg
Mode: leader
```

即 Zookeeper 的配置所在路径及 Mode 属性为 leader，与先前启动时观测到的日志一致，Zookeeper 启动完成且一切正常。

### 6.5.2 启动Kafka集群

本节操作与第 6.5.1 节开头一样，在第 6.4 节中提到 Kafka 所在节点服务器只需要安装 Docker 及 Docker-Compose 环境，Go 语言及 Fabric 环境无需部署，但为了便于项目更加清晰，本书对此步骤操作依旧会在 172.31.159.131、172.31.159.132、172.31.159.133 和 172.31.159.134 四台服务器上建立一个/home/docker/github.com/hyperledger/fabric/aberic 目录。

分别将 docker-kafka1.yaml、docker-kafka2.yaml、docker-kafka3.yaml 和 docker-kafka4.yaml 上传至 172.31.159.131、172.31.159.132、172.31.159.133 及 172.31.159.134 四台服务器的



/home/docker/github.com/hyperledger/fabric/aberic 目录下，如图 6-11 所示。

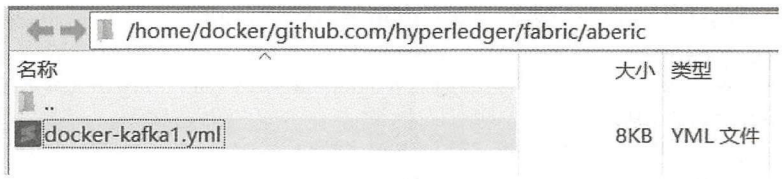


图6-11　Kafka目录

随后在 172.31.159.131 服务器上执行如下命令启动 docker-kafka1.yaml：

```
docker-compose -f docker-kafka1.yml up
```

该命令后面没有加入“-d”参数，是为了直接查看 Kafka 的启动日志。此时读者如果是用测试服务器进行测试操作的话，可能会出现如下提示：

```
    kafka2    | OpenJDK 64-Bit Server VM warning: INFO: os::commit_memory
(0x00000000c0000000, 1073741824, 0) failed; error='Cannot allocate memory' (errno=12)
    kafka2    | #
    kafka2    | # There is insufficient memory for the Java Runtime Environment to
continue.
    kafka2    | # Native memory allocation (mmap) failed to map 1073741824 bytes for
committing reserved memory.
    kafka2    | # An error report file with more information is saved as:
    kafka2    | # //hs_err_pid1.log
    kafka2 exited with code 1
```

由于测试环境的服务器配置可能正好是 1G，而 Kafka 中的 heap-opts 默认是 1G，从而导致内存不足。如是测试需要，可在 Kafka 配置文件中环境变量里加入如下参数：

```
    - KAFKA_HEAP_OPTS=-Xmx256M -Xms128M
```

当 Kafka 启动成功后会有如下日志：

```
    kafka1    | [2018-04-16 05:39:49,712] INFO [TransactionCoordinator id=1] Starting
up. (kafka.coordinator.transaction.TransactionCoordinator)
    kafka1    | [2018-04-16 05:39:49,719] INFO [TransactionCoordinator id=1] Startup
complete. (kafka.coordinator.transaction.TransactionCoordinator)
    kafka1    | [2018-04-16 05:39:49,736] INFO [Transaction Marker Channel Manager 1]:
Starting (kafka.coordinator.transaction.TransactionMarkerChannelManager)
    kafka1    | [2018-04-16 05:39:49,854] INFO Creating /brokers/ids/1 (is it secure?
false) (kafka.utils.ZKCheckedEphemeral)
    kafka1    | [2018-04-16 05:39:49,864] INFO Result of znode creation is: OK
(kafka.utils.ZKCheckedEphemeral)
    kafka1    | [2018-04-16 05:39:49,865] INFO Registered broker 1 at path /brokers/ids
```

```
/1 with addresses: EndPoint(kafka1,9092,ListenerName(PLAINTEXT),PLAINTEXT)
(kafka.utils.ZkUtils)
    kafka1    | [2018-04-16 05:39:49,866] WARN No meta.properties file under dir
/tmp/kafka-logs/meta.properties (kafka.server.BrokerMetadataCheckpoint)
    kafka1    | [2018-04-16 05:39:49,895] INFO Kafka version : 1.0.0 (org.apache.
kafka.common.utils.AppInfoParser)
    kafka1    | [2018-04-16 05:39:49,895] INFO Kafka commitId : aaa7af6d4a11b29d
(org.apache.kafka.common.utils.AppInfoParser)
    kafka1    | [2018-04-16 05:39:49,906] INFO [KafkaServer id=1] started
(kafka.server.KafkaServer)
```

上述日志表示 kafka 已经成功初始化、实例化并启动，接下来按照之前的启动命令依此启动 172.31.159.132、172.31.159.133 及 172.31.159.134 上的 Kafka。

在 Kafka 单个节点服务启动的同时，Zookeeper 的 Leader 节点服务器中的日志会给予反馈信息。

如启动 Kafka1 时，Zookeeper 集群当前 Leader 节点 Zookeeper2 服务器中的日志如下所示：

```
    zookeeper2    | 2018-04-16 06:55:21,883 [myid:2] - INFO
[NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket
connection from /172.31.159.133:41252
    zookeeper2    | 2018-04-16 06:55:21,892 [myid:2] - INFO  [NIOServerCxn. Factory:
0.0.0.0/0.0.0.0:2181:ZooKeeperServer@928] - Client attempting to establish new session
at /172.31.159.133:41252
    zookeeper2    | 2018-04-16 06:55:21,897 [myid:2] - INFO  [SyncThread:2:
FileTxnLog@203] - Creating new log file: log.100000001
    zookeeper2    | 2018-04-16 06:55:21,913 [myid:2] - INFO  [CommitProcessor:2:
ZooKeeperServer@673] - Established session 0x262cd3c250b0000 with negotiated timeout
6000 for client /172.31.159.133:41252
    zookeeper2    | 2018-04-16 06:55:21,973 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0000 type:create cxid:0x5 zxid:0x100000003 txntype:-1
reqpath:n/a Error Path:/brokers Error:KeeperErrorCode = NoNode for /brokers
    zookeeper2    | 2018-04-16 06:55:21,995 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0000 type:create cxid:0xb zxid:0x100000007 txntype:-1
reqpath:n/a Error Path:/config Error:KeeperErrorCode = NoNode for /config
    zookeeper2    | 2018-04-16 06:55:22,020 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0000 type:create cxid:0x13 zxid:0x10000000c txntype:-1
reqpath:n/a Error Path:/admin Error:KeeperErrorCode = NoNode for /admin
    zookeeper2    | 2018-04-16 06:55:22,232 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
```

```
sessionid:0x262cd3c250b0000 type:create cxid:0x1f zxid:0x100000013 txntype:-1
reqpath:n/a Error Path:/cluster Error:KeeperErrorCode = NoNode for /cluster
    zookeeper2    | 2018-04-16 06:55:24,646 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0000 type:setData cxid:0x29 zxid:0x100000017 txntype:-1
reqpath:n/a Error Path:/controller_epoch Error:KeeperErrorCode = NoNode for
/controller_epoch
    zookeeper2    | 2018-04-16 06:55:25,084 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0000 type:delete cxid:0x41 zxid:0x10000001a txntype:-1
reqpath:n/a Error Path:/admin/preferred_replica_election Error:KeeperErrorCode =
NoNode for /admin/preferred_replica_election
    zookeeper2    | 2018-04-16 06:55:25,225 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0000 type:create cxid:0x4b zxid:0x10000001b txntype:-1
reqpath:n/a Error Path:/brokers Error:KeeperErrorCode = NodeExists for /brokers
    zookeeper2    | 2018-04-16 06:55:25,226 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0000 type:create cxid:0x4c zxid:0x10000001c txntype:-1
reqpath:n/a Error Path:/brokers/ids Error:KeeperErrorCode = NodeExists for
/brokers/ids
```

而当 Kafka2、Kafka3 及 Kafka4 启动的时候，Zookeeper 将不再去创建日志文件及相关 ID 信息。这三个节点服务器启动后，在 Zookeeper 的 Leader 节点服务器中的日志内容相似，如下格式：

```
    zookeeper2    | 2018-04-16 06:55:45,109 [myid:2] - INFO  [NIOServerCxn.Factory:
0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket connection from
/172.31.159.132:50544
    zookeeper2    | 2018-04-16 06:55:45,116 [myid:2] - INFO  [NIOServerCxn.Factory:0.
0.0.0/0.0.0.0:2181:ZooKeeperServer@928] - Client attempting to establish new session
at /172.31.159.132:50544
    zookeeper2    | 2018-04-16 06:55:45,119 [myid:2] - INFO  [CommitProcessor:2:
ZooKeeperServer@673] - Established session 0x262cd3c250b0001 with negotiated timeout
6000 for client /172.31.159.132:50544
    zookeeper2    | 2018-04-16 06:55:46,303 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0001 type:create cxid:0x1d zxid:0x100000020 txntype:-1
reqpath:n/a Error Path:/brokers Error:KeeperErrorCode = NodeExists for /brokers
    zookeeper2    | 2018-04-16 06:55:46,303 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0001 type:create cxid:0x1e zxid:0x100000021 txntype:-1
reqpath:n/a Error Path:/brokers/ids Error:KeeperErrorCode = NodeExists for
/brokers/ids
```

至此，Kafka 集群已经完成启动。

### 6.5.3 启动Orderer集群

Orderer 排序服务集群启动首先分别将 docker-orderer0.yaml、docker-orderer1.yaml 及 docker-orderer2.yaml 上传至 172.31.159.130、172.31.143.22 及 172.31.143.23 三台服务器的 /home/docker/github.com/hyperledger/fabric/aberic 目录下。

与此同时，也需要在三台服务器上分别上传在第 6.1.2 节中所生成的 genesis.block 创世区块文件到各自的/home/docker/github.com/hyperledger/fabric/aberic/channel-artifacts 目录下，如果没有 channel-artifacts 目录，则手动创建一个。

由于 Orderer 排序服务的启动已经开始依赖于第 6.1.1 节和第 6.1.2 节中配置文件的创建文件，故需将同时准备好 crypto-config.yaml 配置文件所生成的节点文件，并将 crypto-config 下的 ordererOrganizations 上传至各个服务器的 /home/docker/github.com/hyperledger/fabric/aberic/crypto-config 目录下，如果没有 crypto-config 目录，则手动创建一个。

---

**注意：** 上传至/home/docker/github.com/hyperledger/fabric/aberic/crypto-config 目录下的 ordererOrganizations 文件夹的再下一层有一个 orderers 目录，该目录下有三个文件夹，并非都要上传至每一台 Orderer 排序节点服务器上，只需要将与之名称对应的文件夹上传即可。如 orderer2.example.com 文件夹上传至 Orderer2 排序服务器上即可。

---

随后分别在上述三台服务器上执行如下命令启动 docker-orderer.yaml：

```
docker-compose -f docker-orderer0.yml up
docker-compose -f docker-orderer1.yml up -d
docker-compose -f docker-orderer2.yml up -d
```

通过对 Orderer1 的启动日志进行观察，可以看到如下日志：

```
[orderer/common/server] createLedgerFactory -> DEBU 01e Ledger dir: /var/
hyperledger/production/orderer
[kvledger.util] CreateDirIfMissing -> DEBU 01f CreateDirIfMissing [/var/
hyperledger/production/orderer/index/]
[kvledger.util] logDirStatus -> DEBU 020 Before creating dir - [/var/hyperledger/
production/orderer/index/] does not exist
[kvledger.util] logDirStatus -> DEBU 021 After creating dir - [/var/hyperledger/
production/orderer/index/] exists
[fsblkstorage] newBlockfileMgr -> DEBU 022 newBlockfileMgr() initializing file-
based block storage for ledger: testchainid
[kvledger.util] CreateDirIfMissing -> DEBU 023 CreateDirIfMissing [/var/
```

```
hyperledger/production/orderer/chains/testchainid/]
    [kvledger.util] logDirStatus -> DEBU 024 Before creating dir - [/var/hyperledger/
production/orderer/chains/testchainid/] does not exist
    [kvledger.util] logDirStatus -> DEBU 025 After creating dir - [/var/hyperledger/
production/orderer/chains/testchainid/] exists
    [fsblkstorage] newBlockfileMgr -> INFO 026 Getting block information from block
storage
    [fsblkstorage] constructCheckpointInfoFromBlockFiles -> DEBU 027 Retrieving
checkpoint info from block files
    [fsblkstorage] retrieveLastFileSuffix -> DEBU 028 retrieveLastFileSuffix()
```

在 Orderer 启动的时候，创建了一个名为 testchainid 的系统 Channel。紧接着观察之前尚未关闭的 Kafka 日志，有如下所示：

```
    kafka2    | [2018-04-16 07:01:55,133] INFO Replica loaded for partition testchainid
-0 with initial high watermark 0 (kafka.cluster.Replica)
    kafka2    | [2018-04-16 07:01:55,198] INFO Loading producer state from offset 0 for
partition testchainid-0 with message format version 2 (kafka.log.Log)
    kafka2    | [2018-04-16 07:01:55,221] INFO Completed load of log testchainid-0 with
1 log segments, log start offset 0 and log end offset 0 in 58 ms (kafka.log.Log)
    kafka2    | [2018-04-16 07:01:55,223] INFO Created log for partition [testchainid,0]
in /tmp/kafka-logs with properties {compression.type -> producer, message.format.
version -> 1.0-IV0, file.delete.delay.ms -> 60000, max.message.bytes -> 103809024,
min.compaction.lag.ms -> 0, message.timestamp.type -> CreateTime, min.insync.replicas
-> 2, segment.jitter.ms -> 0, preallocate -> false, min.cleanable.dirty.ratio -> 0.5,
index.interval.bytes -> 4096, unclean.leader.election.enable -> false, retention.bytes
-> -1, delete.retention.ms -> 86400000, cleanup.policy -> [delete], flush.ms ->
9223372036854775807, segment.ms -> 604800000, segment.bytes -> 1073741824,
retention.ms -> -1, message.timestamp.difference.max.ms -> 9223372036854775807,
segment.index.bytes -> 10485760, flush.messages -> 9223372036854775807}.
(kafka.log.LogManager)
    kafka2    | [2018-04-16 07:01:55,224] INFO [Partition testchainid-0 broker=2] No
checkpointed highwatermark is found for partition testchainid-0 (kafka.cluster.
Partition)
    kafka2    | [2018-04-16 07:01:55,228] INFO Replica loaded for partition testchainid
-0 with initial high watermark 0 (kafka.cluster.Replica)
    kafka2    | [2018-04-16 07:01:55,228] INFO Replica loaded for partition testchainid
-0 with initial high watermark 0 (kafka.cluster.Replica)
    kafka2    | [2018-04-16 07:01:55,233] INFO [ReplicaFetcherManager on broker 2]
Removed fetcher for partitions testchainid-0 (kafka.server.ReplicaFetcherManager)
    kafka2    | [2018-04-16 07:01:55,288] INFO [ReplicaFetcherManager on broker 2]
Added fetcher for partitions List([testchainid-0, initOffset 0 to broker
BrokerEndPoint(4,kafka4,9092)] ) (kafka.server.ReplicaFetcherManager)
    kafka2    | [2018-04-16 07:01:55,311] INFO [ReplicaFetcher replicaId=2, leaderId=4,
fetcherId=0] Starting (kafka.server.ReplicaFetcherThread)
```

```
    kafka2    | [2018-04-16 07:01:55,337] WARN [ReplicaFetcher replicaId=2, leaderId=4,
fetcherId=0] Based on follower's leader epoch, leader replied with an unknown offset
in testchainid-0. High watermark 0 will be used for truncation.
(kafka.server.ReplicaFetcherThread)
    kafka2    | [2018-04-16 07:01:55,343] INFO Truncating testchainid-0 to 0 has no
effect as the largest offset in the log is -1. (kafka.log.Log)
    kafka2    | [2018-04-16 07:01:55,373] INFO Updated PartitionLeaderEpoch. New:
{epoch:0, offset:0}, Current: {epoch:-1, offset-1} for Partition: testchainid-0. Cache
now contains 0 entries. (kafka.server.epoch.LeaderEpochFileCache)
```

可以发现，Orderer 启动所创建的系统 Channeltestchainid 已经进入了 Kafka 消息队列中进行处理。

同时也能够在 Zookeeper 的 Leader 节点服务器看到如下所示日志：

```
    zookeeper2    | 2018-04-16 07:01:54,948 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0002 type:setData cxid:0x26 zxid:0x10000002d txntype:-1
reqpath:n/a Error Path:/config/topics/testchainid Error:KeeperErrorCode = NoNode for
/config/topics/testchainid
    zookeeper2    | 2018-04-16 07:01:54,955 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0002 type:create cxid:0x28 zxid:0x10000002e txntype:-1
reqpath:n/a Error Path:/config/topics Error:KeeperErrorCode = NodeExists for
/config/topics
    zookeeper2    | 2018-04-16 07:01:54,991 [myid:2] - INFO  [ProcessThread(sid:2
cport:-1)::PrepRequestProcessor@649] - Got user-level KeeperException when processing
sessionid:0x262cd3c250b0000 type:create cxid:0x6b zxid:0x100000031 txntype:-1
reqpath:n/a Error Path:/brokers/topics/testchainid/partitions/0 Error:KeeperErrorCode
= NoNode for /brokers/topics/testchainid/partitions/0
```

在 Orderer1 和 Orderer2 中的启动日志会有如下所示：

```
    [orderer/consensus/kafka] startThread -> INFO 116 [channel: testchainid] Channel
consumer set up successfully
    [orderer/consensus/kafka] startThread -> INFO 117 [channel: testchainid] Start
phase completed successfully
```

至此，Orderer 排序服务集群已经启动完成。

## 6.6 集群环境测试

由于 Peer 节点的交互止步于 Orderer 排序服务节点，即 Peer 并不关注顶层建设是 Solo 还是 Kafka，所以 Peer 节点的配置文件写法及内容与第 5.3 节类似。

但本章的测试侧重点除了集群部署外，还包含自定义节点配置文件测试。故这里会准备三份配置文件，分别是 peer0.org1、foo27.org2 以及 bar.org2 的节点 YAML 文件。

首先来看 peer0.org1 的配置文件 docker-peer0org1.yaml，源码如下所示：

```
version: '2'

services:

  couchdb:
    container_name: couchdb
    image: hyperledger/fabric-couchdb
    # Comment/Uncomment the port mapping if you want to hide/expose the CouchDB
service,
    # for example map it to utilize Fauxton User Interface in dev environments.
    ports:
      - "5984:5984"

  ca:
    container_name: ca
    image: hyperledger/fabric-ca
    environment:
      - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
      - FABRIC_CA_SERVER_CA_NAME=ca
      - FABRIC_CA_SERVER_TLS_ENABLED=false
      - FABRIC_CA_SERVER_TLS_CERTFILE=/etc/hyperledger/fabric-ca-server-
config/ca.org1.example.com-cert.pem
      - FABRIC_CA_SERVER_TLS_KEYFILE=/etc/hyperledger/fabric-ca-server-
config/3d2104b3f30d26073d0cc5022a63c2d82324e85b712759c2fd99881e5280da6a_sk
    ports:
      - "7054:7054"
    command: sh -c 'fabric-ca-server start --ca.certfile /etc/hyperledger/fabric-
ca-server-config/ca.org1.example.com-cert.pem --ca.keyfile /etc/hyperledger/fabric-ca-
server-config/3d2104b3f30d26073d0cc5022a63c2d82324e85b712759c2fd99881e5280da6a_sk -b
admin:adminpw -d'
    volumes:
      - ./crypto-config/peerOrganizations/org1.example.com/ca/:/etc/hyperledger/
fabric-ca-server-config

  peer0.org1.example.com:
    container_name: peer0.org1.example.com
    image: hyperledger/fabric-peer
    environment:
      - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
      - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb:5984
```

```
        - CORE_PEER_ID=peer0.org1.example.com
        - CORE_PEER_NETWORKID=aberic
        - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
        - CORE_PEER_CHAINCODELISTENADDRESS=peer0.org1.example.com:7052
        - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051
        - CORE_PEER_LOCALMSPID=Org1MSP

        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # the following setting starts chaincode containers on the same
        # bridge network as the peers
        # https://docs.docker.com/compose/networking/
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
        - CORE_PEER_GOSSIP_USELEADERELECTION=true
        - CORE_PEER_GOSSIP_ORGLEADER=false
        - CORE_PEER_PROFILE_ENABLED=false
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
      volumes:
          - /var/run/:/host/var/run/
          - ./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.
example.com/msp:/etc/hyperledger/fabric/msp
          - ./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.
example.com/tls:/etc/hyperledger/fabric/tls
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      command: peer node start
      ports:
        - 7051:7051
        - 7052:7052
        - 7053:7053
      depends_on:
        - couchdb
      networks:
        default:
          aliases:
            - example
      extra_hosts:
       - "orderer0.example.com:172.31.159.130"
       - "orderer1.example.com:172.31.143.22"
```

```
      - "orderer2.example.com:172.31.143.23"

    cli:
      container_name: cli
      image: hyperledger/fabric-tools
      tty: true
      environment:
        - GOPATH=/opt/gopath
        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_PEER_ID=cli
        - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
        - CORE_PEER_LOCALMSPID=Org1MSP
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/server
.crt
        -
CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerO
rganizations/org1.example.com/peers/peer0.org1.example.com/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/
fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls
/ca.crt
        - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer
/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      volumes:
          - /var/run/:/host/var/run/
          - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/chaincode/go
          - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
          - ./scripts:/opt/gopath/src/github.com/hyperledger/fabric/peer/scripts/
          - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/
channel-artifacts
      depends_on:
        - peer0.org1.example.com
      extra_hosts:
       - "orderer0.example.com:172.31.159.130"
       - "orderer1.example.com:172.31.143.22"
       - "orderer2.example.com:172.31.143.23"
       - "peer0.org1.example.com:172.31.159.129"
```

其中，CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE 是用来创建 Docker 容器的参数。容器可以使用 ipam 和 dns-server 来高效地创建集群网络模式——为容器设置网络模

式。支持的标准值是：“host”（默认）、“bridge”、“ipvlan”和“none”。

---

**注意**：CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE 如果按照配置文档介绍的标准值赋值可能会存在问题，比如引发智能合约实例化超时等错误。这里给的值是 aberic_default，属半自定义值，当 Orderer 排序服务容器启动的时候，其 network 会根据当前 YAML 所在目录的名称再加上_default 后缀生成默认的 network，因为当前 YAML 文件所在目录名为 aberic，所以是 aberic_default。

---

CORE_PEER_ID 表示当前 Peer 节点在 Fabric 网络中的 ID，允许对网络进行逻辑分离。

CORE_PEER_ADDRESS 表示当前 Peer 节点在 Fabric 网络中的地址。

CORE_PEER_GOSSIP_EXTERNALENDPOINT 表示所有外部服务访问当前 Peer 节点服务的地址。

CORE_PEER_CHAINCODELISTENADDRESS 表示当前 Peer 节点用来监听入站链码连接的监听地址，端口为 7052。

CORE_PEER_LOCALMSPID 表示当前 Peer 节点服务的成员服务提供者 ID，在本 Peer 节点中是 Org1MSP。

CORE_PEER_GOSSIP_USELEADERELECTION 表示何时 Peer 节点将初始化“Leader”选择的动态算法，其中 Leader 是与 Orderer 排序服务建立连接的 Peer 节点，并使用交付协议从 Orderer 排序服务中提取账本区块。建议在大型的同行网络中使用 true。

CORE_PEER_GOSSIP_ORGLEADER 表示静态地定义当前 Peer 节点为组织的“Leader”，这意味着当前 Peer 节点将保持与 Orderer 排序服务的连接，并在其自己的组织中跨节点地传播区块。

将 docker-peer0org1.yaml 配置文件上传至 172.31.159.129 服务器的/home/docker/github.com/hyperledger/fabric/aberic 目录下，将第 6.1.1 节生成的 mychannel.tx 频道文件上传至/home/docker/github.com/hyperledger/fabric/aberic/channel-artifacts 目录下，将第 6.1.1 节生成的 peerOrganizations 上传至/home/docker/github.com/hyperledger/ fabric/aberic/crypto-config 目录下，且仅上传 org1 相关配置即可。

随后进入 172.31.159.129 服务器并执行如下命令，启动 Peer 节点服务：

```
docker-compose -f docker-peer0org1.yaml up -d
```

启动完成后，执行如下命令创建频道：

```
peer channel create -o orderer0.example.com:7050 -c mychannel -t 50 -f ./channel-artifacts/mychannel.tx
```

频道创建成功后，执行如下命令加入该频道：

```
peer channel join -b mychannel.block
```

加入完成后，执行如下命令安装智能合约：

```
peer chaincode install -n mycc -p github.com/hyperledger/fabric/chaincode/
go/chaincode_example02 -v 1.0
```

智能合约安装完成后，执行如下命令实例化：

```
peer chaincode instantiate -o orderer0.example.com:7050 -C mychannel -n mycc -c
'{"Args":["init","A","10","B","10"]}' -P "OR ('Org1MSP.member','Org2MSP.member')" -v
1.0
```

实例化完成后，执行如下命令查询 A 的值：

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","A"]}'
```

查询后可以得到如下日志：

```
2018-04-17 03:19:43.500 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local
MSP
2018-04-17 03:19:43.500 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining
default signing identity
2018-04-17 03:19:43.500 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003
Using default escc
2018-04-17 03:19:43.500 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004
Using default vscc
2018-04-17 03:19:43.500 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java
chaincode disabled
2018-04-17 03:19:43.500 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext:
0AAB070A6708031A0C08CFCFD5D60510...6D7963631A0A0A0571756572790A0141
2018-04-17 03:19:43.500 UTC [msp/identity] Sign -> DEBU 007 Sign: digest:
1CB8F611AC8F024186D346D9CF88B03B898D25FDCA7C819BC7702E6523E015DB
Query Result: 10
2018-04-17 03:19:43.525 UTC [main] main -> INFO 008 Exiting.....
```

说明 peer0org1 节点已经成功启动，并创建且加入了频道，也安装并实例化了智能合约，经过测试后智能合约一切运行正常。

接下来需要对 foo27.org2 以及 bar.org2 的节点 YAML 文件进行编辑并测试其节点是否能正常运转，这两个节点相对于模板节点较为特殊，属于自定义节点与半自定义节点。

先看 foo27.org2 的配置文件 docker-foo27org2.yaml，源码如下所示：

```
version: '2'
```

```
  services:

    couchdb:
      container_name: couchdb
      image: hyperledger/fabric-couchdb
      # Comment/Uncomment the port mapping if you want to hide/expose the CouchDB
service,
      # for example map it to utilize Fauxton User Interface in dev environments.
      ports:
        - "5984:5984"

    ca:
      container_name: ca
      image: hyperledger/fabric-ca
      environment:
        - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
        - FABRIC_CA_SERVER_CA_NAME=ca
        - FABRIC_CA_SERVER_TLS_ENABLED=false
        - FABRIC_CA_SERVER_TLS_CERTFILE=/etc/hyperledger/fabric-ca-server-config/
ca.org2.example.com-cert.pem
        - FABRIC_CA_SERVER_TLS_KEYFILE=/etc/hyperledger/fabric-ca-server-config/
9cb2c6149cf08af7efc4ddd370aeb6f0e875405d47ffc4ee55e53735082a40f4_sk
      ports:
        - "7054:7054"
      command: sh -c 'fabric-ca-server start --ca.certfile /etc/hyperledger/fabric-
ca-server-config/ca.org2.example.com-cert.pem --ca.keyfile /etc/hyperledger/fabric-ca-
server-config/9cb2c6149cf08af7efc4ddd370aeb6f0e875405d47ffc4ee55e53735082a40f4_sk -b
admin:adminpw -d'
      volumes:
        - ./crypto-config/peerOrganizations/org2.example.com/ca/:/etc/hyperledger
/fabric-ca-server-config

    foo27.org2.example.com:
      container_name: foo27.org2.example.com
      image: hyperledger/fabric-peer
      environment:
        - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
        - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=172.31.143.21:5984

        - CORE_PEER_ID=foo27.org2.example.com
        - CORE_PEER_NETWORKID=aberic
        - CORE_PEER_ADDRESS=foo27.org2.example.com:7051
        - CORE_PEER_CHAINCODEADDRESS=foo27.org2.example.com:7052
        - CORE_PEER_CHAINCODELISTENADDRESS=foo27.org2.example.com:7052
```

```
        - CORE_PEER_GOSSIP_EXTERNALENDPOINT=foo27.org2.example.com:7051
        - CORE_PEER_LOCALMSPID=Org2MSP

        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # the following setting starts chaincode containers on the same
        # bridge network as the peers
        # https://docs.docker.com/compose/networking/
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
        - CORE_VM_DOCKER_TLS_ENABLED=false
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
        - CORE_PEER_GOSSIP_USELEADERELECTION=true
        - CORE_PEER_GOSSIP_ORGLEADER=false
        - CORE_PEER_PROFILE_ENABLED=false
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
      volumes:
          - /var/run/:/host/var/run/
          - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/chaincode/go
          - ./crypto-config/peerOrganizations/org2.example.com/peers/foo27.org2.
example.com/msp:/etc/hyperledger/fabric/msp
          - ./crypto-config/peerOrganizations/org2.example.com/peers/foo27.org2.
example .com/tls:/etc/hyperledger/fabric/tls
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      command: peer node start
      ports:
        - 7051:7051
        - 7052:7052
        - 7053:7053
      depends_on:
        - couchdb
      networks:
        default:
          aliases:
            - aberic
      extra_hosts:
       - "orderer0.example.com:172.31.159.130"
       - "orderer1.example.com:172.31.143.22"
       - "orderer2.example.com:172.31.143.23"

    cli:
```

```
    container_name: cli
    image: hyperledger/fabric-tools
    tty: true
    environment:
      - GOPATH=/opt/gopath
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      # - CORE_LOGGING_LEVEL=ERROR
      - CORE_LOGGING_LEVEL=DEBUG
      - CORE_PEER_ID=cli
      - CORE_PEER_ADDRESS=foo27.org2.example.com:7051
      - CORE_PEER_CHAINCODELISTENADDRESS=foo27.org2.example.com:7052
      - CORE_PEER_LOCALMSPID=Org2MSP
      - CORE_PEER_TLS_ENABLED=false
      - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/peers/foo27.org2.example.com/tls/server.crt
      - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/peers/foo27.org2.example.com/tls/server.key
      - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/peers/foo27.org2.example.com/tls/ca.crt
      - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
    volumes:
        - /var/run/:/host/var/run/
        - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/chaincode/go
        - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
        - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/
channel-artifacts
    depends_on:
      - foo27.org2.example.com
    extra_hosts:
     - "orderer0.example.com:172.31.159.130"
     - "orderer1.example.com:172.31.143.22"
     - "orderer2.example.com:172.31.143.23"
     - "foo27.org2.example.com:172.31.143.21"
```

将 docker-foo27org2.yaml 文件上传至 172.31.143.21 服务器/home/docker/github.com/hyperledger/fabric/aberic 目录下，随后执行如下命令启动容器：

```
docker-compose -f docker-foo27org2.yaml up -d
```

参考第 5.4 节的步骤，将 mychannel.block 文件复制到当前服务器/home/docker/github.com/hyperledger/fabric/aberic/channel-artifacts 目录下，并通过“docker cp”命令将其复制到 cli 容器 peer 目录下。

执行如下命令，加入 mychannel 频道：

```
peer channel join -b mychannel.block
```

加入完成后，执行如下命令安装智能合约

```
peer chaincode install -n mycc -p github.com/hyperledger/fabric/chaincode/
go/chaincode_example02 -v 1.0
```

智能合约安装完成后，执行如下命令查询 A 的值：

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","A"]}'
```

查询后可以得到的日志与 peer0org1 节点服务器一致，说明该自定义配置节点启动及运行完成。

接着看 bar.org2 的配置文件 docker-barorg2.yaml，源码如下所示：

```
version: '2'

services:

  couchdb:
    container_name: couchdb
    image: hyperledger/fabric-couchdb
    # Comment/Uncomment the port mapping if you want to hide/expose the CouchDB
service,
    # for example map it to utilize Fauxton User Interface in dev environments.
    ports:
      - "5984:5984"

  ca:
    container_name: ca
    image: hyperledger/fabric-ca
    environment:
      - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
      - FABRIC_CA_SERVER_CA_NAME=ca
      - FABRIC_CA_SERVER_TLS_ENABLED=false
      - FABRIC_CA_SERVER_TLS_CERTFILE=/etc/hyperledger/fabric-ca-server-
config/ca.org2.example.com-cert.pem
      - FABRIC_CA_SERVER_TLS_KEYFILE=/etc/hyperledger/fabric-ca-server-config/
9cb2c6149cf08af7efc4ddd370aeb6f0e875405d47ffc4ee55e53735082a40f4_sk
    ports:
      - "7054:7054"
    command: sh -c 'fabric-ca-server start --ca.certfile /etc/hyperledger/fabric-
ca-server-config/ca.org2.example.com-cert.pem --ca.keyfile /etc/hyperledger/fabric-ca-
server-config/9cb2c6149cf08af7efc4ddd370aeb6f0e875405d47ffc4ee55e53735082a40f4_sk -b
```

```
admin:adminpw -d'
      volumes:
        - ./crypto-
config/peerOrganizations/org2.example.com/ca/:/etc/hyperledger/fabric-ca-server-config

    bar.org2.example.com:
      container_name: bar.org2.example.com
      image: hyperledger/fabric-peer
      environment:
        - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
        - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=172.31.143.21:5984

        - CORE_PEER_ID=bar.org2.example.com
        - CORE_PEER_NETWORKID=aberic
        - CORE_PEER_ADDRESS=bar.org2.example.com:7051
        - CORE_PEER_CHAINCODEADDRESS=bar.org2.example.com:7052
        - CORE_PEER_CHAINCODELISTENADDRESS=bar.org2.example.com:7052
        - CORE_PEER_GOSSIP_EXTERNALENDPOINT=bar.org2.example.com:7051
        - CORE_PEER_LOCALMSPID=Org2MSP

        - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
        # the following setting starts chaincode containers on the same
        # bridge network as the peers
        # https://docs.docker.com/compose/networking/
        - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
        - CORE_VM_DOCKER_TLS_ENABLED=false
        # - CORE_LOGGING_LEVEL=ERROR
        - CORE_LOGGING_LEVEL=DEBUG
        - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
        - CORE_PEER_GOSSIP_USELEADERELECTION=true
        - CORE_PEER_GOSSIP_ORGLEADER=false
        - CORE_PEER_PROFILE_ENABLED=false
        - CORE_PEER_TLS_ENABLED=false
        - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
        - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
        - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
      volumes:
          - /var/run/:/host/var/run/
          - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/chaincode/go
          - ./crypto-config/peerOrganizations/org2.example.com/peers/bar.org2.
example.com/msp:/etc/hyperledger/fabric/msp
          - ./crypto-config/peerOrganizations/org2.example.com/peers/bar.
org2.example.com/tls:/etc/hyperledger/fabric/tls
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
```

```
    command: peer node start
    ports:
      - 7051:7051
      - 7052:7052
      - 7053:7053
    depends_on:
      - couchdb
    networks:
      default:
        aliases:
          - aberic
    extra_hosts:
     - "orderer0.example.com:172.31.159.130"
     - "orderer1.example.com:172.31.143.22"
     - "orderer2.example.com:172.31.143.23"

  cli:
    container_name: cli
    image: hyperledger/fabric-tools
    tty: true
    environment:
      - GOPATH=/opt/gopath
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      # - CORE_LOGGING_LEVEL=ERROR
      - CORE_LOGGING_LEVEL=DEBUG
      - CORE_PEER_ID=cli
      - CORE_PEER_ADDRESS=bar.org2.example.com:7051
      - CORE_PEER_CHAINCODELISTENADDRESS=bar.org2.example.com:7052
      - CORE_PEER_LOCALMSPID=Org2MSP
      - CORE_PEER_TLS_ENABLED=false
      - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/peers/bar.org2.example.com/tls/server.crt
      - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/
crypto/peerOrganizations/org2.example.com/peers/bar.org2.example.com/tls/server.key
      - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric
/peer/crypto/peerOrganizations/org2.example.com/peers/bar.org2.example.com/tls/ca.crt
      - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer
/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
    volumes:
      - /var/run/:/host/var/run/
      - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/chaincode/go
      - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
      - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/
```

```
channel-artifacts
      depends_on:
        - bar.org2.example.com
      extra_hosts:
       - "orderer0.example.com:172.31.159.130"
       - "orderer1.example.com:172.31.143.22"
       - "orderer2.example.com:172.31.143.23"
       - "bar.org2.example.com:172.31.143.21"
```

通过与 foo27.org2 节点服务器执行相同的测试方案，结果也能顺利地完成频道加入、智能合约安装、查询等步骤。

至此，整个集群环境测试完成，除了一些名称规范及待自定义的部分外，基本已经符合了生产部署的条件。

## 6.7 本章小结

对于 HyperLedger Fabric 网络而言，生产部署必须是建立在 Kafka 集群的基础上。当然，如果业务量实在是太小，且没有联盟的必要性，可以考虑采用 Solo 方式，但需要做好数据备份。在 Orderer 排序服务或 Peer 节点服务宕机的时候，可以通过备份将数据恢复回来，避免造成不必要的损失。

但总体来说，依旧建议采用 Kafka 集群的方式搭建 Fabric 网络。就个人理解，HyperLedger Fabric 严格来说不是一个应用，而是一个平台，它有频道和成员的概念。当搭建好一个基于 Kafka 集群的 Fabric 网络后，可以通过频道来做软隔离，将不同的账本分隔开来，以实现业务分离。通过成员管理来新建准入门槛，也可以保证业务访问的安全，至少在 CA 数据被泄漏的时候，只要不是根 CA 或中间 CA，都能最大化地保证数据的安全性。

本章做了许多关于事物流程、读写集等理论上的介绍，也进行了一次 Kafka 集群上手过程，囊括了 Zookeeper 集群、Kafka 集群、Orderer 排序服务集群，也对普通模板生成的 Peer 节点执行了最基本的操作，实现频道创建、加入，智能合约安装、实例化及执行等，同时对非模板生成的以自定义方式生成的 Peer 节点做了类似的测试。总体内容较多，需要前后结合来看，更加深入地去了解 Fabric 网络的运行轨迹，在遇到问题的时候也会更容易排查。

---

**注意：**本章中所有 YAML 文件源码及所生成的配置文件等，均可在本书前言的“读者服务”中找到并进行下载。

---

# 第 7 章　智能合约

## 7.1　智能合约概述

智能合约是一种被广泛认可并使用的术语，在 HyperLedger Fabric 中被称为链码。

智能合约拥有自己的执行逻辑，在 HyperLedger Fabric 建立频道的特定网络中被采用为业务规则。这些业务主要是对数据进行逻辑处理，对数据的实际应用是各个组织的自定义规则，不建议将真实场景中的业务规则作为统一标准捆绑进智能合约。

智能合约（建议由 Go 语言编写）将会被一个授权的成员安装并实例化到一台 Peer 节点服务上，随后，普通的业务人员可以使用一个执行有 Fabric-SDK 的客户端与 Peer 节点服务进行交互，从而实现对智能合约的调用。

智能合约在事物流程中进行运转，如果一旦被验证且验证的结果集被发送至 Orderer 排序服务，那么其运行结果中的变化将被共享或同步到 Fabric 网络中的所有 Peer 节点，从而改变 World State。

可以分别从智能合约代码开发人员和智能合约在 Fabric 网络中实际操作人员的角度来看待智能合约。通过前者的视角，开发区块链应用程序或解决方案名为智能合约，即如何通过 Go 语言等进行编码从而实现整个智能合约编写的结果。通过后者的视角来看，区块链的智能合约是面向网络运维人员或运营商的，由一个负责管理整个区块链网络的人来执行相关操作，包括利用 HyperLedger Fabric 的 API 来安装、实例化和升级智能合约，但它不会参与到智能合约的编码过程。

鉴于此，后续第 7.3 节和 7.4 节将分别从智能合约使用及智能合约编写两个角度阐述。

而在开始分析智能合约之前，需要结合前六章的内容对背书策略进行一次概要，同时也便于对后续小节的理解。

## 7.2 背书策略

1. 背书策略概要

背书策略用于指导 Peer 如何确定交易是否得到了认可。当一个 Peer 接收到一个事务时，它会调用与事务的智能合约相关联的 VSCC（验证系统智能合约），作为事务验证流程的一部分，以确定交易的有效性。一个交易包含了一个或多个 Peer 背书节点中的背书支持。VSCC 的任务是做出以下决定：

（1）所有的背书都是有效的（也就是说，它们是有效的签名，而不是预期的消息）。

（2）有适当数量的背书。

（3）背书来自预期的一个或多个来源。

背书策略指的是第二和第三点其中的一种方式。

2. CLI中的背书策略语法

在 CLI 中，使用一种简单的布尔表达式语言来表达对主体的背书策略。

一个主体被描述为 MSP，它的任务是验证签名者的身份，以及签名者在 MSP 中所扮演的角色。目前，支持两个角色，即成员和管理员。角色组成，MSP 必须承载 MSP ID，角色是成员和管理员两个字符串之一。例如，一个有效的主体是“Org0.admin”（任何组织的任何管理员）或是 Org1.member（任何组织的成员）。

该语言的语法是：

```
EXPR(E[, E...])
```

EXPR 使用 AND 或者 OR 其中之一作为表达式，E 要么是一个主体（上面描述的是语法），要么是另一个对 EXPR 的嵌套调用。

例如：

（1）AND('Org1.member', 'Org2.member', 'Org3.member')三个主体必须同时背书并认可签名。

（2）OR('Org1.member', 'Org2.member')两个主体中的任意一个背书并认可签名。

（3）OR('Org1.member', AND('Org2.member', 'Org3.member'))主体 1 背书并认可签名或者主体 2 和主体 3 同时背书并认可签名。

3. 为智能合约指定背书策略

使用这种语法，智能合约部署人员可以请求对 Chaincode 的背书在指定的策略上进行验证。注意-默认策略需要一个来自默认 MSP 的成员的签名。如果在实例化智能合约时，在 CLI 中没有指定策略，则使用此方法。

可以在实例化时使用“-P”关键词指定策略，然后执行策略。

例如：

```
peer chaincode instantiate -C <channelid> -n mycc -P "AND('Org1.member',
'Org2.member')"
```

这个命令使用 AND('Org1.member', 'Org2.member')背书策略来部署 chaincode mycc，该策略要求 Org1 和 Org2 的两个成员都必须签署事务。

## 7.3　使用智能合约

### 7.3.1　智能合约是什么

若想能够很好地操作并使用智能合约，首先需要知道智能合约对于使用者来说意味着什么。

智能合约是一个程序，它是使用 Go 语言编写的，最终在 Java 等其他编程语言中实现了指定的接口。智能合约运行在一个被背书 Peer 进程独立出来的、安全的 Docker 容器中。智能合约通过应用程序提交的事务初始化和管理账本状态。

智能合约通常处理被网络成员认可的业务逻辑，因此它被认为是一种“智能合同”。由智能合约创建的状态只作用于该智能合约，而不能通过另一个智能合约直接访问。但是，在同一个网络中，给定适当的权限，智能合约可以调用另一个智能合约来访问它的状态。

在下面的内容中，将通过区块链通信产品应用方案供应商的视角来探索智能合约。将更加关注供应商对于智能合约生命周期的操作；在区块链网络中，打包、安装、实例化和升级智能合约的过程，是智能合约的操作生命周期的一个功能。

### 7.3.2　智能合约的生命周期

Hyperledger Fabric 的 API 支持与区块链网络中的各个节点进行交互——peers、orderers 和 MSPs——它还允许其中一个在支持的背书节点上打包、安装、实例化和升级智能合约。尽管 Hyperledger Fabric 可以用来管理智能合约的生命周期，但它还是提供了特定语言的 SDK 抽象了 Hyperledger Fabric API 的细节，以促进应用程序的开发。另外，可以通过 CLI 直接访问

Hyperledger Fabric API，在此章节中使用 CLI。

官方提供了四个命令来管理一个智能合约的生命周期：package、install、instantiate 和 upgrade。在未来的版本中，为了不用真正地卸载智能合约，官方也正在考虑添加 stop 和 start 命令操作事务来禁用和重新启用智能合约。

在成功安装并实例化了一个智能合约之后，智能合约就处于活跃中（正在运行），并且可以通过调用事务处理事务。

在安装完毕后，也可以在任何时间对智能合约进行升级。

### 7.3.3 Packaging（包）

1. 智能合约包的组成

- 智能合约由智能合约部署规范（ChaincodeDeploymentSpec）或 CDS 定义。CDS 是根据代码和其他属性（如名称和版本）定义的智能合约包。
- 一个可选的实例化策略，它可以在策略上用相同的策略来描述，用于支持和在背书策略中描述。参考第 7.2 节。
- 由“拥有”智能合约的实体的一组签名。

2. 签名的目的

- 为了建立智能合约的所有权。
- 允许对包的内容进行验证；
- 允许检测包是否篡改。

一个 Channel 上的智能合约实例化事务的创建者，是通过智能合约的实例化策略来验证的。

### 7.3.4 创建package（包）

打包智能合约有两种方法。一种是当想要一个智能合约拥有多个所有者时，需要使用多个身份标志为该智能合约签名。这个工作流程需要首先创建一个已签名的智能合约（一个签署的 CDS），然后通过序列的方式将其传递给其他所有者来签署。

更简单的工作流程是正在发行安装事务的节点的身份签名时部署已签署的 CDS。

首先，将处理更复杂的情况。但是，如果不需要担心多个所有者，那么可以跳过下面的安装智能合约部分。

要创建一个已签名的智能合约包，请使用以下命令：

```
peer chaincode package -n mycc -p github.com/hyperledger/fabric/examples/
```

```
chaincode/go/chaincode_example02 -v 0 -s -S -i "AND('OrgA.admin')" ccpack.out
```

-s 参数是指可以创建一个由多个所有者签署的包，而不是简单地创建一个未处理成修饰过的 CDS。当指定了-s 时，如果其他所有者需要签名，也必须指定-s 参数。否则，这个进程会创建一个除了 CDS 实例化策略之外的已签署 CDS。

-S 参数使用在 core.yaml 中由 LocalMspid 属性值标志的 MSP 来指示该程序的签名。

-S 参数是可选的。但是，如果一个包是在没有签名的情况下创建的，那么它就不能由任何其他所有者使用 signpackage 命令来签署。

-i 参数是可选的，即指定智能合约实例化策略。实例化策略与背书策略具有相同的格式，并指定哪些 ID 可以实例化智能合约。在上面的示例中，只允许使用 OrgA 的 admin 实例化链代码。如果没有提供策略，则使用默认策略，这将只允许 peer 中 MSP 的 admin 身份来实例化智能合约。

### 7.3.5　包签名（Package signing）

一个已经被签名的智能合约包在被创建时候可以交由其他所有者检查并签名，这个工作流程支持智能合约包带外签署。

ChaincodeDeploymentSpec 视需要也许会由共同的全部所有者签名，去创建一个 SignedChaincodeDeploymentSpec（或 SignedCDS）。

---

**注意：**ChaincodeDeploymentSpec 参考官方 GitHub 链接地址 https://github.com/hyperledger/fabric/blob/master/protos/peer/chaincode.proto#L78。SignedChaincodeDeploymentSpec 参考官方 GitHub 链接地址 https://github.com/hyperledger/fabric/blob/master/protos/peer/signed_cc_dep_spec.proto#L26。

---

SignedCDS 包含了 3 个元素：

（1）CDS 包含了智能合约的源码、名称和版本信息。

（2）智能合约的实例化策略，即表示背书策略。

（3）智能合约的所有者列表，以背书的方式定义。

---

**注意：**当智能合约在某些 Channel 上实例化时，此背书策略是由传输层协议使用的带外数据（Out Of Band，OOB）决定的，以提供适当的 MSP 原则。如果没有指定实例化策略，则默认策略是 Channel 的任何 MSP 的 admin。

---

每个所有者通过将其与所有者的身份（例如证书）相结合，并签署结合后的结果来为

ChaincodeDeploymentSpec 背书。

一个智能合约所有者可以使用下面的命令来签署一个以前创建的签名包：

```
peer chaincode signpackage ccpack.out signedccpack.out
```

ccpack.out 和 signedccpack.out 分别是输入包和输出包。

signedccpack.out 包含了使用本地 MSP 签名的包的附加签名。

### 7.3.6 安装智能合约

安装事务将智能合约的源代码打包成一种指定的格式，称为 ChaincodeDeploymentSpec（智能合约部署规范或 CDS），并将其安装到运行该智能合约的 Peer 节点上。

---

**注意：** 必须在 Channel 中的每个背书节点上安装智能合约，以运行智能合约。

---

当安装 API 被简单地给出一个 ChaincodeDeploymentSpec 时，它将默认实例化策略，并包含一个空的所有者列表。

---

**注意：** 智能合约只应该安装在对智能合约拥有的成员的背书 Peer 节点上，以保护网络中其他成员的智能合约逻辑的机密性。那些没有智能合约的成员，不能成为智能合约交易的背书人；也就是说，他们不能执行智能合约。但是，他们仍然可以验证并将事务提交到账本上。

---

要安装一个智能合约，请将一个签署的提案发送到 System Chaincode（系统智能合约）其中被描述为生命周期系统智能合约（Lifecycle System chaincode，LSCC）的部分。例如，要安装使用 CLI 的简单资产智能合约中描述的 sacc 示例智能合约，该命令如下所示：

```
peer chaincode install -n asset_mgmt -v 1.0 -p sacc
```

CLI 容器内执行创建的 SignedChaincodeDeploymentSpec sacc，并将其发送给本地 Peer，本地 Peer 会调用 LSCC 上的安装方法。对-p 选项的参数指定了智能合约的路径，它必须位于用户的 GOPATH 的源码树中，例如$GOPATH/src/sacc。有关命令选项的完整描述，后面将会讲到。

要注意一点，为了在 Peer 上安装，签署的提案的签名必须是 Peer 的本地 MSP 管理员的一个签名。

### 7.3.7 智能合约实例化

实例化事务调用生命周期系统智能合约来创建和初始化一个 Channel 上的智能合约。这是

一个 Chaincode-Channel 绑定过程：智能合约可以绑定到任意数量的 Channel，并分别在每个 Channel 上独立操作。换句话说，不管智能合约安装和实例化了多少个其他 Channel，状态都被隔离到一个事务提交的 Channel 上。

实例化事务的创建者必须满足在 SignedCDS 中包含的智能合约的实例化策略，并且该创建者作为创建该 Channel 配置信息的一部分，也必须是 Channel 上的一个写入者。这对于 Channel 的安全性来说是非常重要的，它可以防止恶意实体部署智能合约或欺骗成员在一个未绑定的 Channel 上执行智能合约。

例如，默认的实例化策略是任何 Channel 上的 MSP 管理员，因此一个智能合约实例化事务的创建者必须是 Channel 管理员的成员。当事务提案到达背书人（节点）的时候，它将验证创建者的签名与实例化策略。并且在将其提交给账本之前，在事务验证期间再次执行此操作。

实例化事务还为 Channel 上的智能合约设置了背书策略。背书策略描述了交易结果的认证要求，被该 Channel 的所有成员所接受。

例如，使用 CLI 实例化 sacc 智能合约，并使用 john 和 0 初始化状态，命令将如下所示：

```
peer chaincode instantiate -n sacc -v 1.0 -c '{"Args":["john","0"]}' -P "OR
('Org1.member','Org2.member')"
```

---

**注意**：签注策略（CLI 使用波兰表示法），它需要来自 Org1 或 Org2 的任意成员的支持，以支持所有的事务到 sacc。也就是说，无论是 Org1 或 Org2 都必须签署在 sacc 上执行调用的结果，以使事务是有效的。

波兰表示法（Polish notation，或波兰记法），是一种逻辑、算术和代数表示方法，其特点是运算符置于操作数的前面，因此也称做前缀表示法。如果运算符的元数（arity）是固定的，则语法上不需要括号仍然能被无歧义地解析。波兰记法是波兰数学家扬·武卡谢维奇 1920 年代引入的，用于简化命题逻辑。

---

在智能合约成功实例化之后，智能合约在 Channel 上进入活跃状态，并准备处理任何背书事务类型支持的事务协议。这些事务是并发处理的，因为它们到达了背书 Peer。

### 7.3.8　升级智能合约

任何时候，智能合约都可以通过更改其版本来进行升级，这是 SignedCDS 的一部分。其他部分，例如所有者和实例化策略是可选的。但是，智能合约的名称必须是相同的，否则它将被视为完全不同的智能合约。

在升级之前，必须将智能合约的新版本安装在需要的背书 Peer 上。升级是一个类似于实例化事务的事务，它将智能合约的新版本绑定到 Channel。其他 Channels 所绑定的旧版本智能

合约将会继续运行旧版本智能合约。换句话说，升级事务只会一次影响一个 Channel，即提交事务的 Channel。

---

注意：由于智能合约的多个版本可能同时处于活跃状态，所以升级过程不会自动删除旧版本，因此用户必须暂时管理这个版本。

---

与实例化事务有一个微妙的区别：升级事务是根据当前的智能合约实例化策略检查的，而不是新策略（如果指定的话）。这是为了确保在当前的实例化策略中指定的现有成员可以升级智能合约。

---

注意：在升级过程中，调用 chaincode Init 函数来执行任何与数据相关的更新或重新初始化它，因此在升级智能合约时必须注意避免重新设置状态。

---

### 7.3.9 停止及启动智能合约

停止和启动智能合约的生命周期事务还没有实现。但是，可以通过从每个背书人中删除智能合约容器和 SignedCDS 包来手动停止智能合约。这是通过在背书 Peer 节点运行的每个主机或虚拟机上删除智能合约的容器来完成的，然后从每个背书 Peer 节点上删除 SignedCDS。

---

注意：官方 TODO-为了从 Peer 节点删除 CDS，首先需要进入 Peer 节点的容器。暂时并没有提供一个能够执行此功能的实用程序脚本。

---

```
docker rm -f <container id>
rm /var/hyperledger/production/chaincodes/<ccname>:<ccversion>
```

Stop 在工作流程中是有用的，可以在控制方式上进行升级。在进行升级之前，可以在所有 Peer 上停止一个智能合约。

### 7.3.10 CLI（客户端）

官方正在评估为 Hyperledger Fabric Peer 二进制文件分发特定平台的二进制文件的需求。目前，可以简单地从运行的 Docker 容器中调用命令。

要查看当前可用的 CLI 命令，请在运行的 fabric-peer Docker 容器中执行以下命令：

```
docker run -it hyperledger/fabric-peer bash
# peer chaincode --help
```

**注意：** 可使用 docker exec -it cli（容器名） bash 命令进入 cli。

这显示了与下面示例相似的输出：

```
Usage:
  peer chaincode [command]

Available Commands:
  install     Package the specified chaincode into a deployment spec and save it on
the peer's path.
  instantiate Deploy the specified chaincode to the network.
  invoke      Invoke the specified chaincode.
  list        Get the instantiated chaincodes on a channel or installed chaincodes
on a peer.
  package     Package the specified chaincode into a deployment spec.
  query       Query using the specified chaincode.
  signpackage Sign the specified chaincode package
  upgrade     Upgrade chaincode.

Flags:
    --cafile string      Path to file containing PEM-encoded trusted certificate(s)
for the ordering endpoint
-h, --help               help for chaincode
-o, --orderer string     Ordering service endpoint
    --tls                Use TLS when communicating with the orderer endpoint
    --transient string   Transient map of arguments in JSON encoding
```

全局 flags：--logging-level string

默认的日志记录 level 和 overrides，参见 core.yaml 的完整语法。

```
-test.coverprofile string Done (default “coverage.cov”)
-v, -version
```

使用“peer chaincode [command] －help”获取更多关于命令的信息。

为了方便在脚本化的应用程序中使用，peer 命令总是在发生命令失败时生成非零返回代码。

智能合约命令的例子：

```
peer chaincode install -n mycc -v 0 -p path/to/my/chaincode/v0
peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a", "b", "c"]}' -C mychannel
peer chaincode install -n mycc -v 1 -p path/to/my/chaincode/v1
peer chaincode upgrade -n mycc -v 1 -c '{"Args":["d", "e", "f"]}' -C mychannel
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","e"]}'
peer chaincode invoke -o orderer.example.com:7050 --tls --cafile $ORDERER_CA -C
mychannel -n mycc -c '{"Args":["invoke","a","b","10"]}'
```

### 7.3.11 系统智能合约

系统智能合约具有相同的编程模型，除了它在 Peer 进程中运行，而不是像普通的智能合约那样在一个单独的容器中运行。因此，系统智能合约被构建到 Peer 的可执行文件中，并且不遵循上面描述的相同的生命周期。特别是安装、实例化和升级并不适用于系统智能合约。

系统智能合约是为了在 Peer 和智能合约之间减少 gRPC 的通信成本，并权衡管理的灵活性。例如，系统智能合约只能用 Peer 二进制进行升级。它还必须注册一个固定的参数集，并且没有背书策略或背书策略功能。

系统智能合约用于 Hyperledger Fabric 以实现许多系统行为，使它们可以被系统集成商所取代或修改。

当前的系统智能合约列表：

（1）LSCC（Lifecycle System Chaincode）：生命周期系统智能合约处理上面描述的生命周期请求。

（2）CSCC（Configuration System Chaincode）：配置系统智能合约在 Peer 端处理 Channel 配置。

（3）QSCC（Query System Chaincode）：查询系统智能合约提供了分类查询 API，例如获取块和事务。

（4）ESCC（Endorsement System Chaincode）：背书系统智能合约通过签署事务提案响应来处理支持。

（5）VSCC（Validation System Chaincode）：验证系统智能合约处理事务验证，包括检查背书策略和多版本并发控制。

在修改或替换这些系统智能合约时必须注意，特别是 LSCC、ESCC 和 VSCC，因为它们在主事务执行路径中。值得注意的是，当 VSCC 在将其提交到账本之前验证一个块，重要的是，Channel 中的所有 Peer 节点都要计算相同的验证，以避免账本差异（非确定性）。因此，如果 VSCC 被修改或替换，就需要特别地处理和维护。

## 7.4 编写智能合约

### 7.4.1 开发人员眼中的智能合约

智能合约通常处理由网络成员同意的业务逻辑，因此它类似于“智能合同”。可以调用智能合约来更新或查询提案事务中的账本。在适当的许可下，智能合约可以调用另一个智能合约，在相同的 Channel 或不同的 Channel 中，以访问它的状态。如果被调用的智能合约与调用

的智能合约在不同的 Channel 上，则只允许读取查询。也就是说，在不同的 Channel 上被调用的智能合约只是一个查询，它不参与后续提交阶段的状态验证检查。

在后续的内容中，将通过应用程序开发人员的视角来探索智能合约。首先将介绍一个简单的智能合约示例应用程序，并在智能合约 Shim API 中遍历每种方法的用途。

### 7.4.2　智能合约接口

每个智能合约程序都必须实现 Chaincode 接口。

---

**注意：** Go 语言实现的可以参考官方接口地址 https://godoc.org/github.com/ hyperledger/fabric/core/chaincode/shim#Chaincode；Node.js 语言实现的可以参考官方接口地址 https://fabric-shim.github.io/ChaincodeInterface.html。

---

该接口方法被调用来响应接收的事务。特别是当智能合约接收 instantiate（实例化）或 upgrade（升级）事务时，将调用 Init 方法，以便智能合约可以执行任何必要的初始化，包括应用程序状态的初始化。Invoke 方法被调用，通过响应收到的一个 invoke 事务处理交易内容。

智能合约“shim”API 的另一个接口是 ChaincodeStubInterface。

---

**注意：** Go 语言实现的可以参考官方接口地址为 https://godoc.org/ github.com/hyperledger/fabric/core/chaincode/shim#ChaincodeStub；Node.js 语言实现的可以参考官方接口地址为 https://fabric-shim.github.io/ChaincodeStub.html。

---

该接口方法用于访问和修改网络账本，并在智能合约之间进行调用。

下面将通过官方文档提供的素材实现一个简单的智能合约应用程序，以便演示这些 API 的使用，该应用程序做一个简单的“资产”管理。

### 7.4.3　一个简单的资产智能合约

#### 1. 智能合约样板准备

如果服务器还没有安装好 Go 语言环境，请参考第 1.4 节进行安装及配置。

首先需要准备好必备的 Go 编程样板，与所有的智能合约一样，它实现了 Chaincode 接口，特别是实现了 Init 和 Invoke 方法。因此，需要给智能合约添加 import 语句，以获得必要的依赖项。这里将导入 chaincode shim 包和 peer protobuf 包。接下来，继续添加一个名为 SimpleAsset 的 struct 作为 Chaincode shim 方法的接收方。

具体示例代码如下：

```
package main

import (
    "fmt"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

// SimpleAsset实现一个简单的智能合约来管理资产。
type SimpleAsset struct {
}
```

2. 智能合约初始化方法

接下来，将要继续实现 Init 方法。示例如下：

```
// Init在智能合约实例化过程中被调用初始化任何数据。
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {

}
```

**注意**：智能合约升级也调用这个函数。在编写将要升级的现有版本的智能合约时，请确保适当地修改 Init 函数。特别是，如果没有“迁移”或者没有什么东西可以作为升级的一部分进行初始化，那么就提供一个空的“Init”方法。

接下来，将使用 ChaincodeStubInterface.GetStringArgs 方法检索 Init 所需要的参数，并核实其参数数量的有效性。在这个例子中，需要传入的是一个键值对，即两个参数。

```
// Init在智能合约实例化过程中被调用来初始化任何数据。
// 注意，智能合约升级也调用这个方法来重置或迁移数据，
// 所以要小心避免在无意中破坏了账本数据的情况！
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
  // 从交易请求中获取请求参数数组。
  args := stub.GetStringArgs()
  if len(args) != 2 {
    return shim.Error("Incorrect arguments. Expecting a key and a value")
  }
}
```

接下来，既然已经确定了调用是否具备有效性，那么在调用时传入参数有效的情况下，将

把初始状态存储在账本中。

为了达到上述目的，这里将调用 ChaincodeStubInterface.PutState 作为参数传递的键和值。

假设一切顺利，返回一个 Peer 响应对象。表示初始化的操作是成功的。具体示例代码如下：

```
// Init在智能合约实例化过程中被调用来初始化任何数据。
// 注意，智能合约升级也调用这个方法来重置或迁移数据，
// 所以要小心避免在无意中破坏了账本数据的情况！
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
  // 从交易请求中获取请求参数数组。
  args := stub.GetStringArgs()
  if len(args) != 2 {
    return shim.Error("Incorrect arguments. Expecting a key and a value")
  }

  // 在这里通过调用stub.PutState()方法来存入资产信息。

  // 将资产的value和相关联key存入账本。
  err := stub.PutState(args[0], []byte(args[1]))
  if err != nil {
    return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
  }
  return shim.Success(nil)
}
```

3. 调用智能合约

首先，需要添加 Invoke 方法的样板。

```
// 在智能合约上的每笔交易都会调用Invoke。
// 每一个事务都是由Init方法创建的资产的“get”或“set”。
// “set”方法可以通过指定新的键-值对来创建新资产。
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {

}
```

与上面的 Init 方法一样，这里需要从 ChaincodeStubInterface 接口中提取参数。Invoke 方法的参数是将要调用的智能合约应用程序方法的名称。在这个例子中，所编写的应用程序只有两个功能：set 和 get，它允许设定一个资产的值，或者检索它的当前状态。

首先调用 ChaincodeStubInterface.GetFunctionAndParameters 方法，提取方法名和该智能合约应用程序功能的参数。具体示例如下：

```
// 在智能合约上的每笔交易都会调用Invoke。
```

```
// 每一个事务都是由Init方法创建的资产的“get”或“set”。
// “set”方法可以通过指定新的键-值对来创建新资产。
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // 从交易请求中获取方法名和请求参数数组。
    fn, args := stub.GetFunctionAndParameters()

}
```

接下来，将继续验证方法名是否为 set 或 get，并调用那些智能合约应用程序的方法，通过 shim.Success 或 shim.Error 方法返回适当的响应，这些方法将响应序列化成 gRPC protobuf 消息。具体示例代码如下：

```
// 在智能合约上的每笔交易都会调用Invoke。
// 每一个事务都是由Init方法创建的资产的“get”或“set”。
// “set”方法可以通过指定新的键-值对来创建新资产。
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // 从交易请求中获取方法名和请求参数数组。
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
            result, err = set(stub, args)
    } else {
            result, err = get(stub, args)
    }
    if err != nil {
            return shim.Error(err.Error())
    }

    // 返回成功结果。
    return shim.Success([]byte(result))
}
```

4. 实现Chaincode应用程序

如之前所述，目前编写的智能合约应用程序实现了两个可以通过 Invoke 方法调用的方法。现在则开始一步步实现这些功能。这里需要注意一下，正如上面提到的，为了访问账本的状态，这里将利用智能合约 shim API 中的 ChaincodeStubInterface.PutState 和 ChaincodeStubInterface.GetState 方法。

```
// 将资产（包括key和value）存储在账本上。
// 如果key存在，它会用新key覆盖value。
func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
```

```
    if len(args) != 2 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
    }

    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
        return "", fmt.Errorf("Failed to set asset: %s", args[0])
    }
    return args[1], nil
}

// 获取指定资产key的value。
func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    if len(args) != 1 {
        return "", fmt.Errorf("Incorrect arguments. Expecting a key")
    }

    value, err := stub.GetState(args[0])
    if err != nil {
        return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0],
err)
    }
    if value == nil {
        return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}
```

5. 完整版合约

最后，需要添加一个 main 方法，它将调用 shim.Start 方法。这是整个智能合约应用程序的入口方法。具体源码示例如下：

```
package main

import (
    "fmt"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)

// SimpleAsset实现一个简单的智能合约来管理资产。
type SimpleAsset struct {
}
```

```
// Init在智能合约实例化过程中被调用来初始化任何数据。
// 注意，智能合约升级也调用这个方法来重置或迁移数据，
// 所以要小心避免在无意中破坏了账本数据的情况！
func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // Get the args from the transaction proposal
    args := stub.GetStringArgs()
    if len(args) != 2 {
           return shim.Error("Incorrect arguments. Expecting a key and a value")
    }

    // 在这里通过调用stub.PutState()方法来存入资产信息

    // 将资产的value和相关联key存入账本。
    err := stub.PutState(args[0], []byte(args[1]))
    if err != nil {
           return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
    }
    return shim.Success(nil)
}

// 在智能合约上的每笔交易都会调用Invoke。
// 每一个事务都是由Init方法创建的资产的“get”或“set”。
// “set”方法可以通过指定新的键-值对来创建新资产。
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    // 从交易请求中获取方法名和请求参数数组。
    fn, args := stub.GetFunctionAndParameters()

    var result string
    var err error
    if fn == "set" {
           result, err = set(stub, args)
    } else { // assume 'get' even if fn is nil
           result, err = get(stub, args)
    }
    if err != nil {
           return shim.Error(err.Error())
    }

    // 返回成功结果。
    return shim.Success([]byte(result))
}

// 将资产（包括key和value）存储在账本上。
```

```
    // 如果key存在，它会用新key覆盖value。
    func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
        if len(args) != 2 {
                return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
        }

        err := stub.PutState(args[0], []byte(args[1]))
        if err != nil {
                return "", fmt.Errorf("Failed to set asset: %s", args[0])
        }
        return args[1], nil
    }

    // 获取指定资产key的value。
    func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
        if len(args) != 1 {
                return "", fmt.Errorf("Incorrect arguments. Expecting a key")
        }

        value, err := stub.GetState(args[0])
        if err != nil {
                return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0],
err)
        }
        if value == nil {
                return "", fmt.Errorf("Asset not found: %s", args[0])
        }
        return string(value), nil
    }

    // main方法在实例化过程中启动容器中的智能合约。
    func main() {
        if err := shim.Start(new(SimpleAsset)); err != nil {
                fmt.Printf("Error starting SimpleAsset chaincode: %s", err)
        }
    }
```

### 6. 构建智能合约环境

这一步开始编译之前所写的智能合约代码。

具体执行如下命令：

```
    go get -u github.com/hyperledger/fabric/core/chaincode/shim
    go build
```

如果出现报错，则表示智能合约中代码存在逻辑错误或语法错误。假定上述没有错误，可以继续下一步，测试已经编写好的智能合约。

7. 使用开发模式测试

一般情况下，智能合约是由 Peer 节点启动和维护的，但是在“dev 模式”中，智能合约是由用户自己构建和启动的。这种模式在智能合约开发阶段周期中进行快速开发、构建、运行、调试非常有用。

接下来介绍如何开启“dev 模式”，利用预先生成的 Orderer 和 channel artifacts 进行示例开发网络。因此，用户可以立即跳转到编译智能合约和驱动调用的过程。

8. 安装Hyperledger Fabric Samples

在服务器上确定任意一个位置用来放置 Hyperledger Fabric Samples 仓库，并在终端窗口中进入该目录。下面的命令将执行以下步骤：

步骤 1：如果需要，克隆 hyperledger/fabric-samples 仓库。

步骤 2：切换合适的版本标签。

步骤 3：为指定到 fabric-samples 仓库根目录的版本安装 Hyperledger Fabric 特定于平台的二进制文件和配置文件（具体可参考第 3.1 节）。

步骤 4：下载用于指定版本的 Hyperledger Fabric 的 Docker 镜像（具体可参考第 2.4.2 节）。

如果上述工作都准备完成了，并且在通过终端进入了将要安装 Fabric Samples 和二进制文件的目录中，继续执行下面的命令：

```
curl -sSL https://goo.gl/6wtTN5 | bash -s 1.1.0
```

**注意**：如果在上面的 curl 命令中遇到了一个错误，那么可能是 curl 版本太低了，它不能处理重定向或不受环境的支持。可以进入 curl 官网，地址为 https://curl.haxx.se/download.html，找到符合服务器环境的最新版的 curl 并下载安装。另外，可以将上述地址替换未缩短的链接 https://github.com/hyperledger/fabric/blob/master/scripts/boot.sh。

**注意**：可以使用上面的命令来获得任何已发布版本的 Hyperledger Fabric。只需将“1.1.0”替换为想要安装的版本的版本标志符。

上面的命令下载并执行一个 bash 脚本，它将下载并提取需要设置 Fabric 网络的所有特定于平台的二进制文件，并将它们放入上面创建的克隆仓库中。它检索六个特定于平台的二进制文件：Cryptogen、configtxgen、configtxlator、peer、orderer 以及 fabric-ca-client。

并将它们放置在当前工作目录的 bin 子目录中。

也可以将其添加到 PATH 环境变量中，这样就可以在不完全限定每一个二进制文件路径的情况下获得这些变量。例如执行如下命令：

```
export PATH=<path to download location>/bin:$PATH
```

最后，该脚本将把 Docker Hub 中的 Docker 镜像从 Docker Hub 下载到本地并将其标记为“latest”。

---

**注意**：关于这些特定的平台二进制文件，在第 3.1 节中也提到了离线获取办法，关于 Docker 镜像的处理在第 2.4.2 节中有提及。

---

9. 测试智能合约

接下来，通过终端进入到克隆到本地的 fabric-samples 仓库的 chaincode-docker-devmode 目录下。

执行如下命令：

```
docker-compose -f docker-compose-simple.yaml up
```

上面启动的网络使用单点式的 orderer 配置文件启动，并在“dev 模式”中启动了 Peer 节点。它还会启动两个额外的容器——一个用于智能合约环境和一个与智能合约交互的 CLI。创建和联接 Channel 的命令嵌入到 CLI 容器中，因此可以立即跳转到智能合约启动调用。

接下来继续执行如下命令：

```
docker exec -it chaincode bash
```

可以看到类似如下命令入口：

```
root@d2629980e76b:/opt/gopath/src/chaincode#
```

现在就可以编译智能合约了，执行如下命令：

```
cd sacc
go build
```

执行如下命令，开始运行智能合约：

```
CORE_PEER_ADDRESS=peer:7052 CORE_CHAINCODE_ID_NAME=mycc:0 ./sacc
```

当 Peer 节点成功注册了该智能合约，则会显示智能合约相关的日志信息。但在这个阶段，智能合约还没有与任何 Channel 有关联，需要在随后的智能合约实例化命令过后才开始关联。

即使目前处于--peer-chaincodedev 模式，但仍然需要安装智能合约，以便生命周期系统智能合约能够正常地通过它的检查。官方提示：在以后的版本中，这一步操作可能不再需要，毕竟是在开发模式中。

这里将利用 CLI 容器来驱动这些调用，这些操作在之前的章节中都有讲述，这里再次指出，执行如下命令进入 CLI 容器：

```
docker exec -it cli bash
```

继续执行如下命令，安装及实例化智能合约：

```
peer chaincode install -p chaincodedev/chaincode/sacc -n mycc -v 0
peer chaincode instantiate -n mycc -v 0 -c '{"Args":["a","10"]}' -C myc
```

接下来发出一个 invoke 来改变“a”的值到“20”，执行如下命令：

```
peer chaincode invoke -n mycc -c '{"Args":["set", "a", "20"]}' -C myc
```

最后，查询 a。应该看到值为 20，执行如下命令：

```
peer chaincode query -n mycc -c '{"Args":["query","a"]}' -C myc
```

在默认情况下，只安装并实例化了 sacc。但是，可以通过将它们添加到智能合约子目录并重新启动这个网络来轻松测试不同的智能合约。在这一点上，它们将在智能合约容器中访问。

## 7.5 加密智能合约

在一些特定的场景中，有可能需要用到将与 key 关联的 value 全部或部分地加密。例如，如果一个人的身份证号码或地址被写入到账本中，那么这些数据应该不会也不应该以明文形式出现。智能合约加密是通过利用“实体扩展（Entities Extension）”来实现的，该扩展是一个带有工厂方法功能的 BCCSP（区块链密码服务提供者）包装器，可以执行加密操作，如椭圆曲线数字签名进行加密。打个比方，如果需要进行加密，智能合约的调用者通过“瞬态字段（Transient Field：短暂的成员变量意味着，不会贯穿对象的序列化和反序列化，只会存在当前，是短暂的存在而非持久的存在）”传递密码密钥。然后，同样的密钥可以用于后续的查询操作，允许对加密状态值进行适当的解密。具体可以参考 Encc 示例。

**注意：**Entities Extension 可以参考官方地址 https://github.com/hyperledger/fabric/tree/master/core/chaincode/shim/ext/entities。有关更多信息和示例，可以参见 Encc 示例，地址为 https://github.com/hyperledger/fabric/tree/master/examples/chaincode/go/enccc_example。

在 Encc 示例中，需要特别关注 utils.go 工具类，这个工具类加载了智能合约 shim API 和实体扩展，并构建了一个新的函数类（例如加密用 encryptAndPutState 和解密用 getStateAndDecrypt），然后可以启用智能合约加密。因此，智能合约现在可以与基本的 shim API 结合使用，并添加加密和解密的附加功能。

既然加密智能合约这样场景必不可少，那么接下来开始介绍如何使用 EncCC。

要测试 EncCC，需要首先生成一个 AES 256 位 key 使用 base64 编码的字符串，这样它可以作为 JSON 传递给 Peer 节点智能合约调用的瞬态参数。

在开始之前，必须使用 govendor 添加外部依赖项。进入“enccc_example”文件目录中执行以下命令：

```
govendor init
govendor add +external
```

接下来生成加密和解密密钥。该示例将模拟共享密钥，以便密钥用于加密和解密：

```
ENCKEY=`openssl rand 32 -base64` && DECKEY=$ENCKEY
```

此时，可以调用智能合约加密键-值对，如下：

**注意：**下面操作假定智能合约已经被实例化，并且 Peer 节点服务也已经加入了名为“my-ch”的 Channel。

```
peer chaincode invoke -n enccc -C my-ch -c '{"Args":["ENCRYPT","key1","value1"]}' -
transient "{\"ENCKEY\":\"$ENCKEY\"}"
```

这个调用将使用一个随机的 IV（用来和密钥组合成密钥种子）进行加密。如果智能合约调用需要被多个 Peer 节点支持，那么这样的操作是不可取的，因为它会导致各自背书中的读（写）集相互冲突。可以通过指定 IV 进行确定性加密，执行如下命令，首先必须创建一个 IV：

```
IV=`openssl rand 16 -base64`
```

然后，IV 可以在瞬态字段中指定，如下命令所示：

```
peer chaincode invoke -n enccc -C my-ch -c '{"Args":["ENCRYPT","key2","value2"]}' --
transient "{\"ENCKEY\":\"$ENCKEY\",\"IV\":\"$IV\"}"
```

两个这样的调用将产生相等的KVS写入结果，它可以被多个节点所支持。

该值可以按如下方式检索：

```
peer chaincode query -n enccc -C my-ch -c '{"Args":["DECRYPT","key1"]}' -transient
"{\"DECKEY\":\"$DECKEY\"}"

peer chaincode query -n enccc -C my-ch -c '{"Args":["DECRYPT","key2"]}' -transient
"{\"DECKEY\":\"$DECKEY\"}"
```

**注意**：在上述示例中，使用链码查询操作，虽然瞬态字段的使用保证了内容不会被写入到账本中，但是智能合约解密消息并将其放入请求响应中。调用会将结果保存在所有Channel读取器的账本中，而查询可以被丢弃，因此结果仍然是机密的。

为了测试签名并验证，还需要用恰当曲线生成ECDSA密钥，如下所示：

```
On Intel:
SIGKEY=`openssl ecparam -name prime256v1 -genkey | tail -n5 | base64 -w0` &&
VERKEY=$SIGKEY

On Mac:
SIGKEY=`openssl ecparam -name prime256v1 -genkey | tail -n5 | base64` && VERKEY= $SIGKEY
```

在这一点上，可以调用智能合约来签署，然后对键-值对进行加密，如下所示：

```
peer chaincode invoke -n enccc -C my-ch -c
'{"Args":["ENCRYPTSIGN","key3","value3"]}
' --transient "{\"ENCKEY\":\"$ENCKEY\",\"SIGKEY\":\"$SIGKEY\"}"
```

和前述类似，执行如下命令可以进行检索：

```
peer chaincode query -n enccc -C my-ch -c '{"Args":["DECRYPTVERIFY","key3"]}' --
transient "{\"DECKEY\":\"$DECKEY\",\"VERKEY\":\"$VERKEY\"}"
```

至此，智能合约的加密内容已介绍完毕。

## 7.6 系统合约插件

系统智能合约是专门的智能合约，作为Peer节点服务进程的一部分运行，而不是在单独的Docker容器中运行的用户智能合约。因此，它们可以在Peer节点上获得更多的资源，并且可以用于实现那些难以或不可能通过用户智能合约实现的特性。系统智能合约的例子是ESCC（背书系统智能合约），用于支持建议；QSCC（查询系统智能合约），用于账本和其他与

Fabric 相关的查询；VSCC（验证系统智能合约），用于在提交时进行验证事务。

与用户智能合约不同的是，系统智能合约没有安装并使用 SDKs 或 CLI 的方案进行实例化。它是由 Peer 节点服务在启动时注册和部署的。

系统智能合约可以通过两种方式链接到 Peer 节点：静态和动态地使用 Go 插件。本节将概述如何开发和加载系统智能合约作为插件。

### 1. 开发插件

系统智能合约是一个用 Go 插件包编写的程序。

一个插件包含一个带有导出符号的“main package”，并使用命令“go build -buildmode=plugin”进行构建。

每个系统智能合约都必须实现 Chaincode 接口，并导出一个构造器方法，该方法与“main package”中的签名“func New() shim.Chaincode”相匹配。

可以在“examples/plugin/scc”的仓库中找到一个示例。

现有的智能合约，如 QSCC，也可以作为某些特性的模板，例如访问控制，通常是通过系统智能合约实现的。现有的系统智能合约还可以作为日志和测试等方面的最佳实践的参考。

---

**注意：**在导入的包中，Go 标准库要求插件必须包含与主机应用程序相同的导入包（在示例中是 fabric）。

---

### 2. 配置插件

插件是在 core.yaml 的 chaincode.systemPlugin 部分配置的，如下所示：

```
chaincode:
systemPlugins:
- enabled: true
  name: mysyscc
  path: /opt/lib/syscc.so
  invokableExternal: true
  invokableCC2CC: true
```

在 core.yaml 的 chaincode.system 部分中还必须对系统智能合约进行允许处理：

```
chaincode:
system:
mysyscc: enable
```

至此，系统智能合约插件的相关知识已介绍完毕。

## 7.7 智能合约API

本节是关于在实际编写智能合约过程中的一些接口方法。

在第 7.4 节中已经详尽介绍了 Init 和 Invoke 作为 Chaincode 接口入口的方法并加上了注释，这里不在本节中进行赘述，下面直接开始介绍官方提供的在编码过程中会用到的接口。

在实际编码过程中，用到得最多的是由 ChaincodeStubInterface 接口提供的方法，该接口提供了各种对数据的增删查以及对系统信息的查询功能方法，见表 7-1。

表 7-1 智能合约方法

| 方 法 | 描 述 |
| --- | --- |
| GetArgs() [][]byte | 返回为智能合约 Init 的参数，并作为字节数组的数组调用 |
| GetStringArgs() []string | 返回用于智能合约 Init 的参数，并作为字符串数组调用。只有当客户端传递打算用作字符串的参数时，才使用 GetStringArgs |
| GetFunctionAndParameters() (string, []string) | 将第一个参数作为方法名并将其余参数作为字符串数组中的参数返回。只有当客户端传递打算用作字符串的参数时才使用 GetFunctionAndParameters |
| GetArgsSlice() ([]byte, error) | 返回用于智能合约 Init 的参数，并作为字节数组调用 |
| GetTxID() string | 返回事务请求的 txid，每笔交易和每个客户端都是独一无二的 |
| GetChannelID() string | 返回请求发送给智能合约来处理的 Channel。这是交易请求的 ChannelID，除非智能合约在不同的 Channel 上调用另一个 Channel |
| InvokeChaincode(chaincodeName string, args [][]byte, channel string) pb.Response | 在本地调用指定的智能合约“Invoke”，使用相同的事务上下文；也就是说，智能合约调用智能合约不会创建一个新的事务消息如果被调用的智能合约位于同一个 Channel 上，那么它只会将被调用的智能合约的读集和写集添加到调用事务中。如果被调用的智能合约位于不同的 Channel 上，则只有响应被返回给调用的智能合约；任何来自智能合约的 PutState 调用都不会对账本产生任何影响；也就是说，在不同的 Channel 上被调用的智能合约将不会有它的读集和写集应用于事务。只有调用智能合约的读集和写集将被应用到事务中。在不同的 Channel 上触发调用智能合约是一个“查询”，它不参与后续提交阶段的状态验证检查。如果“Channel”是空的，则假定是调用者的 Channel |
| GetState(key string) ([]byte, error) | 从账本中返回指定的“key”的值。请注意，GetState 不读取读集的数据，而后者并没有提交给账本。换句话说，GetState 不考虑未经提交的 PutState 修改的数据。如果密钥在状态数据库中不存在，就会返回（nil，nil） |
| PutState(key string, value []byte) error | 将指定的“key”和“value”放入事务的读集作为数据写入建议。PutState 在事务被验证和成功提交之前不会影响账本。简单的 key 不能是空字符串，不能以 null 字符（0x00）开头，以免与复合键的范围查询冲突，后者在内部以 0x00 作为复合键名称空间前缀 |

续表

| 方　法 | 描　述 |
| --- | --- |
| DelState(key string) error | 记录在交易请求的文字中删除指定的“key”。当交易被验证并成功提交时，“key”及其值将被从账本中删除 |
| GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error) | 在账本上的一组键上返回一个范围迭代器。迭代器可以用来迭代 startKey 和 endKey 之间的所有键。这些键由迭代器按词法顺序返回。请注意，startKey 和 endKey 可以是空字符串，这意味着开始或结束时的无限制范围查询。完成后，在返回的 statequeryatorinterface 对象上调用 Close（）。查询在验证阶段被重新执行，以确保结果集自事务背书以来没有发生变化 |
| GetStateByPartialCompositeKey(objectType string, keys []string) (StateQueryIteratorInterface, error) | 根据给定的部分组合键查询账本中的状态。这个方法返回一个迭代器，它可以用来迭代所有的组合键，这些键的前缀与给定的部分复合键匹配。“objectType”和属性应该是只有有效的 utf8 字符串，不应该包含 U+0000 (nil byte)和 U+10FFFF（最大和未分配的代码点）。参见相关方法 SplitCompositeKey 和 CreateCompositeKey。完成后，在返回的 StateQueryIteratorInterface 对象上调用 Close（）。查询在验证阶段被重新执行，以确保结果集自事务背书以来没有发生变化 |
| CreateCompositeKey(objectType string, attributes []string) (string, error) | 将给定的“属性”组合成一个复合键。objectType 和属性应该是只有有效的 utf8 字符串，不应该包含 U+0000 (nil byte)和 U+10FFFF（最大和未分配的代码点）。由此产生的复合键可以用作 PutState（）中的键 |
| SplitCompositeKey(compositeKey string) (string, []string, error) | 将指定的键分割成组合键形成的属性。因此，在范围查询或部分复合键查询中发现的复合键，可以被分割成它们的复合部分 |
| GetQueryResult(query string) (StateQueryIteratorInterface, error) | 对状态数据库执行“富”查询。它只支持有富查询功能的状态数据库，即 CouchDB。查询字符串符合状态数据库自身的语法。返回的迭代器可以用来在查询结果集中迭代，在验证阶段不会重新执行查询，不会检测到幻读，也就是说，其他提交的事务可能已经添加、更新或删除了影响结果集的键，并且在验证或提交时不会检测到这一点。因此，易受此影响的应用程序不应该使用 GetQueryResult 作为更新账簿事务的一部分，并且应该限制对只读智能合约操作的使用 |
| GetHistoryForKey(key string) (HistoryQueryIteratorInterface, error) | 返回贯穿时间的键值的历史记录。对于每一个历史键更新，都会返回历史值和相关的交易 ID 和时间戳。时间戳是请求事务头部信息中客户端提供的时间戳。GetHistoryForKey 请求 Peer 节点配置 core.ledger.history.enableHistoryDatabase 是正确的。在验证阶段，查询不会被重新执行，因此不会检测到幻读，也就是说，其他提交的事务可能同时更新了 key，影响了结果集，并且在验证或提交时间内不会被检测到。因此，易受此影响的应用程序不应该使用 GetHistoryForKey 作为更新账本的事务的一部分，并且应该限制对只 |

续表

| 方　法 | 描　述 |
| --- | --- |
| | 读智能合约操作的使用 |
| GetCreator() ([]byte, error) | 返回“SignatureHeader.Creator”的创建者（例如身份）。这是提交交易的代理（或用户）的身份 |
| GetTransient() (map[string][]byte, error) | 返回“ChaincodeProposalPayload.Transient”字段。它是一个包含数据（如加密材料）的映射，它可能被用来实现某种形式的应用程序级机密性。该字段的内容，应该总是在事务中被忽略，并被排除在账本之外 |
| GetBinding() ([]byte, error) | 返回交易绑定，它用于在应用程序数据（如上面的瞬态字段中存储的）之间强制链接到请求本身。这对于避免可能的重放是有用的 |
| GetDecorations() map[string][]byte | 返回来自 Peer 节点请求的附加数据（如果适用的话）。这些数据由 Peer 节点的 decorator 设置，后者对智能合约进行追加或改变传递输入给智能合约 |
| GetSignedProposal() (*pb.SignedProposal, error) | 返回 SignedProposal 对象，它包含事务请求的所有数据元素 |
| GetTxTimestamp() (*timestamp.Timestamp, error) | 返回交易创建时的时间戳。这是从事务 ChannelHeader 中获取的，因此它将指示客户端的时间戳，并在所有背书人之间具有相同的值 |
| SetEvent(name string, payload []byte) error | 允许智能合约在对请求的响应中设置一个事件，作为交易的一部分。无论交易的有效性如何，该事件都将在事务中可用 |

## 7.8 Peer节点与合智能约

Peer 节点服务中的智能合约相关命令，允许管理员在 Peer 节点上执行智能合约相关操作，如安装、实例化、调用、打包、查询和升级智能合约。

Peer 节点操作智能合约有如下命令：

```
peer chaincode install     [flags]
peer chaincode instantiate  [flags]
peer chaincode invoke      [flags]
peer chaincode list        [flags]
peer chaincode package     [flags]
peer chaincode query       [flags]
peer chaincode signpackage  [flags]
peer chaincode upgrade     [flags]
```

不同的子命令选项（安装、实例化等）与 Peer 节点相关的不同的智能合约操作相关。例如，使用 Peer 节点对智能合约安装子命令选项在 Peer 节点上安装合约，或者通过 Peer 节点对智能合约查询子命令选项查询 Peer 节点账本上的当前值的合约。

在本节中，将描述每个 Peer 节点中智能合约的命令及其选项。

每个 Peer 节点中智能合约的命令都有一组特定于单个子命令的标志，以及一组与所有 Peer 节点中智能合约子命令相关的全局标志。并不是所有的子命令都使用这些标志。例如，查询子命令不需要 orderer 标志。

每个标志都用相关的子命令描述，全局命令的标志见表 7-2。

表 7-2　全局命令的标志

| 命　令 | 描　述 |
| --- | --- |
| --cafile <string> | 用于 Orderer 排序服务节点的包含 PEM-encoded 的一个或多个可信证书的路径 |
| --certfile <string> | 包含 PEM-encoded 的 X.509 公钥的路径到与 orderer 排序服务节点的相互 TLS 通信的路径 |
| --keyfile <string> | 包含 PEM-encoded 的私钥的路径到与 orderer 排序服务节点的相互 TLS 通信的路径 |
| -o or --orderer <string> | 执行 Orderer 排序服务节点地址为<hostname or IP address>:<port>格式 |
| --ordererTLSHostnameOverride <string> | 当验证使用 TLS 连接到 Orderer 排序服务节点时，主机名将被使用 |
| --tls | 在与 orderer 排序服务节点通信时使用 TLS |
| --transient <string> | JSON 编码中的参数的瞬态映射 |
| --logging-level <string> | 默认日志级别的覆盖方案，参见 core.yaml 中的完整语法 |

## 7.8.1　安装智能合约

Peer 节点中智能合约的安装命令允许管理员将智能合约安装到 Peer 节点的文件系统上。

Peer 节点中安装智能合约语法如下所示：

```
peer chaincode install [flags]
```

安装也可以使用通过 Peer 节点中智能合约的打包命令来执行（请参阅后续的 Peer 节点中智能合约的打包部分）。使用 Peer 节点中智能合约的打包的语法如下：

```
peer chaincode install [chaincode-package-file]
```

chaincode-package-file 是从 Peer 节点中智能合约的打包命令中输出的文件。

在 Peer 节点中安装智能合约的命令有表 7-3 所示特定的标志。

表 7-3　安装命令

| 命　令 | 描　述 |
| --- | --- |
| -c, --ctor <string> | JSON 格式的智能合约的构造函数消息（默认“{}”） |

续表

| 命　令 | 描　述 |
| --- | --- |
| -l, --lang <string> | 指定智能合约编写语言，默认为 Go 语言 |
| -n, --name <string> | 正在安装的智能合约的名称，它可能由字符、破折号和下划线组成 |
| -p, --path <string> | 正在安装的智能合约的路径。对于 Golang（-l Golang）的智能合约，这是相对于 GOPATH 的路径。对于 Node js（-l node）的智能合约，这是绝对路径或者是执行安装命令的相对路径 |
| -v, --version <string> | 正在安装的智能合约的版本。它可以由字符、破折号、下划线、句号和加号组成 |

下面是一些 Peer 节点中安装智能合约的安装命令示例：

如在版本 1.0 中安装名为 mycc 的链码：

```
    peer chaincode install -n mycc -v 1.0 -p github.com/hyperledger/fabric/examples/
chaincode/go/example02/cmd


    .
    .
    .
    2018-02-22 16:33:52.998 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003
Using default escc
    2018-02-22 16:33:52.998 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004
Using default vscc
    .
    .
    .
    2018-02-22 16:33:53.194 UTC [chaincodeCmd] install -> DEBU 010 Installed remotely
response:<status:200 payload:"OK" >
    2018-02-22 16:33:53.194 UTC [main] main -> INFO 011 Exiting.....
```

下面是安装成功的日志消息：

```
    install -> DEBU 010 Installed remotely response:<status:200 payload:"OK" >
```

安装带有“package”命令的智能合约包 ccpack.out：

```
    peer chaincode install ccpack.out


    .
    .
    2018-02-22 18:18:05.584 UTC [chaincodeCmd] install -> DEBU 005 Installed remotely
response:<status:200 payload:"OK" >
```

```
2018-02-22 18:18:05.584 UTC [main] main -> INFO 006 Exiting.....
```

下面是安装成功的日志消息：

```
install -> DEBU 005 Installed remotely response:<status:200 payload:"OK" >
```

接下来开始实例化智能合约。

## 7.8.2　实例化智能合约

在 Peer 节点为成员的 Channel 上，Peer 节点中智能合约的实例化命令允许管理员实例化智能合约。

Peer 节点中智能合约的实例化命令为以下语法。

```
peer chaincode instantiate [flags]
```

Peer 节点中实例化智能合约的命令有表 7-4 所示特定的标志：

表 7-4　实例化智能合约的命令

| 命　令 | 描　述 |
| --- | --- |
| -C, --channelID <string> | 智能合约应该被实例化为所在的 Channel 的名称 |
| -c, --ctor <string> | JSON 格式的智能合约的构造函数消息（默认“{}”） |
| -E, --escc <string> | 用于此智能合约（默认的“escc”）的背书系统合约的名称 |
| -n, --name <string> | 正在安装的智能合约的名称。它可能由字符、破折号和下划线组成 |
| -P, --policy <string> | 与此智能合约相关的背书策略。默认情况下，Fabric 将生成一个类似于“当前 Channel 中组织中的任何成员”的背书策略 |
| -v, --version <string> | 正在安装的智能合约的版本。它可以由字符、破折号、下划线、句号和加号组成 |
| -V, --vscc <string> | 用于此智能合约的验证系统合约的名称（默认的“vscc”） |

同时也支持全局的 Peer 节点命令：

- --cafile <string>
- --certfile <string>
- --keyfile <string>
- -o, --orderer <string>
- --ordererTLSHostnameOverride <string>
- --tls
- --transient <string>

---

**注意：**如果没有指定“orderer”标志，则命令将尝试在发出实例化命令之前，从 Peer 节

点中检索信道的 orderer 信息。

下面是 Peer 节点中智能合约的实例化命令的一些例子，它在名为 mychannel 的 Channel 中实例化了 V1.0 版本的名为 mycc 的智能合约。

使用了 TLS 和 cafile 全局标志，在带有 TLS 的网络中实例化智能合约：

```
    export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
orderer
    Organizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.exampl
e.com-cert.pem
    peer chaincode instantiate -o orderer.example.com:7050 --tls --cafile $ORDERER_CA
-C mychannel -n mycc -v 1.0 -c '{"Args":["init","a","100","b","200"]}' -P "OR
('Org1MSP.peer','Org2MSP.peer')"

    2018-02-22 16:33:53.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001
Using default escc
    2018-02-22 16:33:53.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002
Using default vscc
    2018-02-22 16:34:08.698 UTC [main] main -> INFO 003 Exiting.....
```

只使用特定于命令的选项，来实例化不带有 TLS 的网络中的智能合约：

```
    peer chaincode instantiate -o orderer.example.com:7050 -C mychannel -n mycc -v 1.0
-
    c '{"Args":["init","a","100","b","200"]}' -P "OR
('Org1MSP.peer','Org2MSP.peer')"


    2018-02-22 16:34:09.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001
Using default escc
    2018-02-22 16:34:09.324 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002
Using default vscc
    2018-02-22 16:34:24.698 UTC [main] main -> INFO 003 Exiting.....
```

接下来开始调用智能合约。

### 7.8.3 调用智能合约

Peer 节点中智能合约的调用命令允许管理员使用提供的参数在 Peer 节点上调用智能合约中约定的方法。CLI 通过向 Peer 节点发送交易请求来调用智能合约。Peer 节点将执行智能合约，并将所认可的请求响应（或错误）发送给 CLI。在收到一个已认可的请求响应时，CLI 将使用它构造一个事务并将其发送给 Orderer 排序服务。

Peer 节点中智能合约的调用命令为以下语法：

```
peer chaincode invoke [flags]
```

Peer 节点中调用智能合约的命令有 7-5 所示特定的标志。

表 7-5　调用智能合约的命令

| 命　令 | 描　述 |
| --- | --- |
| -C, --channelID <string> | 智能合约应该被实例化为所在的 Channel 的名称 |
| -c, --ctor <string> | JSON 格式的智能合约的构造函数消息（默认“{}”） |
| -n, --name <string> | 正在安装的智能合约的名称，它可能由字符、破折号和下划线组成 |

-C 同时也支持全局的 Peer 节点命令：

- --cafile <string>
- --certfile <string>
- --keyfile <string>
- -o, --orderer <string>
- --ordererTLSHostnameOverride <string>
- --tls
- --transient <string>

**注意：**如果没有指定“orderer”标志，则命令将尝试在发出实例化命令之前，从 Peer 节点中检索信道的 orderer 信息。

这里有一个 Peer 节点中智能合约的调用命令的例子，它调用 mycc 上的 v1.0 版本的 mycc，请求将 10 个单元从变量 a 移动到变量 b。

```
    peer chaincode invoke -o orderer.example.com:7050 -C mychannel -n mycc -c '{"Args":
    ["invoke","a","b","10"]}'

    2018-02-22 16:34:27.069 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001
Using default escc
    2018-02-22 16:34:27.069 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002
Using default vscc
    .
    .
    .
    2018-02-22 16:34:27.106 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 00a ESCC
invoke result: version:1 response:<status:200 message:"OK" > payload:"\n \237mM\376?
[\214\002 \332\204\035\275q\227\2132A\n\204&\2106\037W|\346#\3413\274\022Y\ nE\022\
024\n\004lscc\022\014\n\n\n\004mycc\022\002\010\003\022-\n\004mycc\022%\n\007\n\001a
```

```
\022\002\010\003\n\007\n\001b\022\002\010\003\032\007\n\001a\032\00290\032\010\n\001b\
032\003210\032\003\010\310\001\"\013\022\004mycc\032\0031.0" endorsement:<endorser:"
\n\007Org1MSP\022\262\006-----BEGIN CERTIFICATE-----
\nMIICLjCCAdWgAwIBAgIRAJYomxY2cqHA/fbRnH5a/bwwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEz
ARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcTDVNhbiBG\ncmFuY2lzY28xGTAXBgNVBAoTEG9yZzEuZXhhbX
BsZS5jb20xHDAaBgNVBAMTE2Nh\nLm9yZzEuZXhhbXBsZS5jb20wHhcNMTgwMjIyMTYyODE0WhcNMjgwMjIwMT
YyODE0\nWjBwMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMN\nU2FuIEZyYW5j
aXNjbzETMBEGA1UECxMKRmFicmljUGVlcjEfMB0GA1UEAxMWcGVl\ncjAub3JnMS5leGFtcGxlLmNvbTBZMBMG
ByqGSM49AgEGCCqGSM49AwEHA0IABDEa\nWNNniN3qOCQL89BGWfY39f5V3o1pi//7JFDHATJXtLgJhkK5KosD
dHuKLYbCqvge\n46u3AC16MZyJRvKBiw6jTTBLMA4GA1UdDwEB/wQEAwIHgDAMBgNVHRMBAf8EAjAA\nMCsGA1
UdIwQkMCKAIN7dJR9dimkFtkus0R5pAOlRz5SA3FB5t8Eaxl9A7lkgMAoG\nCCqGSM49BAMCA0cAMEQCIC2DAs
O9QZzQmKi8OOKwcCh9Gd01YmWIN3oVmaCRr8C7\nAiAlQffq2JFlbh6OWURGOko6RckizG8oVOldZG/Xj3C8lA
==\n-----END CERTIFICATE-----\n" signature:"0D\002 \022_\342\350\344\231G&\237\n \244\
375\302J\220l\302\345\210\335D\250y\253P\0214:\221e\332@\002 \000\254\361\224\247\210\
214L\277\370\222\213\217\301\r\341v\227\265\277\336\256^\217\336\005y*\321\023\025\367
" >
    2018-02-22 16:34:27.107 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00b
Chaincode invoke successful. result: status:200
    2018-02-22 16:34:27
```

接下来可以看到调用日志消息是成功提交的：

```
    chaincodeInvokeOrQuery -> INFO 00b Chaincode invoke successful. result: status:200
```

注意：一个成功的响应表明事务被提交到 Orderer 排序服务节点。然后，交易将被添加到一个区块中，最后由 Channel 上的每个 Peer 节点验证或失效。

接下来开始列出智能合约。

### 7.8.4 列出智能合约

Peer 节点中智能合约的列表命令允许管理员列出安装在 Peer 节点上的智能合约，或者在 Peer 节点为成员的 Channel 上实例化的智能合约。

Peer 节点中智能合约的列表命令为以下语法：

```
    peer chaincode list [--installed|--instantiated -C <channel-name>]
```

Peer 节点中智能合约的实例化命令有表 7-6 所示特定的标志。

表 7-6 列出智能合约的实例化命令

| 命 令 | 描 述 |
| --- | --- |
| -C, --channelID <string> | 智能合约应该被实例化为所在的 Channel 的名称 |
| --installed | 使用这个标志在 Peer 节点上列出已安装的智能合约 |

续表

| 命　令 | 描　述 |
| --- | --- |
| --instantiated | 使用这个标志在一个 Channel 上列出实例化的智能合约，而 Peer 节点是该 Channel 的成员 |

下面是一些 Peer 节点中智能合约的列表命令的例子。

使用“--installed”的标志列出安装在 Peer 节点上的智能合约：

```
    peer chaincode list -installed

Get installed chaincodes on peer:
    Name: mycc, Version: 1.0, Path: github.com/hyperledger/fabric/examples/chaincode/
go/example02/cmd, Id: 8cc2730fdafd0b28ef734eac12b29df5fc98ad98bdb1b7e0ef96265c3d893d61
2018-02-22 17:07:13.476 UTC [main] main -> INFO 001 Exiting.....
```

可以看到，在 Peer 节点安装了一个名为 mycc 的智能合约，它的版本号为 1.0。

使用--instantiated 与-C（ChannelID）标志相结合，列出在 Channel 上实例化的智能合约：

```
    peer chaincode list --instantiated -C mychannel

    Get instantiated chaincodes on channel mychannel:
    Name: mycc, Version: 1.0, Path: github.com/hyperledger/fabric/examples/chaincode
/go/example02/cmd, Escc: escc, Vscc: vscc
    2018-02-22 17:07:42.969 UTC [main] main -> INFO 001 Exiting.....
```

可以看到版本号为 v1.0 的智能合约 mycc 在名为 mychannel 的 Channel 被实例化了。

接下来开始打包智能合约。

## 7.8.5　打包智能合约

“peer chaincode package”命令允许管理员打包执行智能合约安装所必需的内容。确保相同的智能合约包可以在多个 Peer 节点上安装的一致性。

Peer 节点中智能合约的打包命令为以下语法：

```
    peer chaincode package [output-file] [flags]
```

Peer 节点中智能合约的打包命令有表 7-7 所示特定的标志。

表 7-7　打包智能合约的命令

| 命　令 | 描　述 |
| --- | --- |
| -c, --ctor <string> | JSON 格式的智能合约的构造函数消息（默认“{}”） |
| -i, --instantiate-policy <string> | 智能合约的实例化策略。目前只支持最需要 1 个签名的策略（例如，“OR (‘Org1MSP.peer’,’Org2MSP.peer’)”） |

续表

| 命　令 | 描　述 |
| --- | --- |
| -l, --lang <string> | 指定智能合约编写语言，默认为 Go 语言 |
| -n, --name <string> | 正在安装的智能合约的名称，它可能由字符、破折号和下划线组成 |
| -P, --policy <string> | 与此智能合约相关的背书策略。在默认情况下，Fabric 将生成一个类似于“当前 Channel 中组织中的任何成员”的背书策略 |
| -s, --cc-package | 除了原始的智能合约部署规范之外，也可以创建一个用于存储智能合约所有权信息的包，可参考下面注释 |
| -S, --sign | 使用-s 标志，指定此标志以使用本地 MSP（可参考下面注释）来添加所有者对包的支持 |
| -v, --version <string> | 正在安装的智能合约的版本，它可以由字符、破折号、下划线、句号和加号组成 |

**注意：**“-s”和“-S”命令的元数据目前还没有使用。这些命令是为了将来的扩展而设计的，并且很可能会进行实现更改。建议不使用它们。

这是一个 Peer 节点中智能合约的打包命令的例子，它在 1.1 版本中打包了名为 mycc 的智能合约，创建了智能合约部署规范，使用本地 MSP 来签署包，并将其输出为 ccpack.out：

```
   peer chaincode package ccpack.out -n mycc -p
github.com/hyperledger/fabric/examples/chaincode/go/example02/cmd -v 1.1 -s -S
    .
    .
    .
    2018-02-22 17:27:01.404 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003
Using default escc
    2018-02-22 17:27:01.405 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004
Using default vscc
    .
    .
    .
    2018-02-22 17:27:01.879 UTC [chaincodeCmd] chaincodePackage -> DEBU 011 Packaged
chaincode into deployment spec of size <3426>, with args = [ccpack.out]
    2018-02-22 17:27:01.879 UTC [main] main -> INFO 012 Exiting.....
```

接下来开始查询智能合约。

### 7.8.6 查询智能合约

Peer 节点中智能合约的查询命令允许通过调用智能合约上的 Invoke 方法来查询智能合约。query 和 invoke 子命令之间的区别在于，在成功的响应中，invoke 所得将交易提交给

Orderer 排序服务节点，而 query 只输出响应，按照规定的输出标准输出成功或其他结果。

Peer 节点中智能合约的查询命令为以下语法：

```
peer chaincode query [flags]
```

Peer 节点中智能合约的查询命令有表 7-8 所示特定的标志。

表 7-8　查询智能合约的命令

| 命　令 | 描　述 |
| --- | --- |
| -C, --channelID <string> | 智能合约应该被实例化为所在的 Channel 的名称 |
| -c, --ctor <string> | JSON 格式的智能合约的构造函数消息（默认“{}”） |
| -n, --name <string> | 正在安装的智能合约的名称，它可能由字符、破折号和下划线组成 |
| -r –raw | 将查询值输出为原始字节（默认） |
| -x --hex | 在十六进制中输出查询值字节数组 |

同时也支持全局的 Peer 节点命令：

- --transient <string>

这里有一个 Peer 节点中智能合约的查询命令的例子，它查询在 v1.0 中名为 mycc 的智能合约的 Peer 节点的账本，以获得变量 a 的值：

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'

2018-02-22 16:34:30.816 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 001
Using default escc
2018-02-22 16:34:30.816 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 002
Using default vscc
Query Result: 90
```

可以从输出中看到变量 a 在查询时的值为 90。

接下来开始签名智能合约包。

## 7.8.7　签名智能合约包

Peer 节点中智能合约的 signpackage 命令向给定的智能合约包添加签名，创建使用-s 和-S 选项和“peer chaincode signpackage”命令。

“peer chaincode signpackage”命令为以下语法：

```
peer chaincode signpackage <inputpackage> <outputpackage>
```

这是一个 Peer 节点中智能合约的 signpackage 命令的例子，它接受一个已有的签名包，并创建一个带有本地 MSP 的签名的新软件包。

```
peer chaincode signpackage ccwith1sig.pak ccwith2sig.pak
```

```
Wrote signed package to ccwith2sig.pak successfully
2018-02-24 19:32:47.189 EST [main] main -> INFO 002 Exiting.....
```

接下来开始升级智能合约。

### 7.8.8 升级智能合约

Peer 节点中智能合约的升级命令允许管理员将在 Channel 上已经实例化过的智能合约升级到一个新版本。

Peer 节点中智能合约的命令为以下语法：

```
peer chaincode upgrade [flags]
```

Peer 节点中智能合约的升级命令有表 7-9 所示特定的标志。

表 7-9 升级智能合约的命令

| 命　令 | 描　述 |
| --- | --- |
| -C, --channelID <string> | 智能合约应该被实例化为所在的 Channel 的名称 |
| -c, --ctor <string> | JSON 格式的智能合约的构造函数消息（默认“{}”） |
| -E, --escc <string> | 用于此智能合约（默认的“escc”）的背书系统合约的名称 |
| -n, --name <string> | 正在安装的智能合约的名称，它可能由字符、破折号和下划线组成 |
| -P, --policy <string> | 与此智能合约相关的背书策略。在默认情况下，Fabric 将生成一个类似于“当前 Channel 中组织中的任何成员”的背书策略 |
| -v, --version <string> | 正在安装的智能合约的版本，它可以由字符、破折号、下划线、句号和加号组成 |
| -V, --vscc <string> | 用于此智能合约的验证系统合约的名称（默认的“vscc”） |

同时也支持全局的 Peer 节点命令：

- --cafile <string>
- -o, --orderer <string>
- --tls

**注意**：如果没有指定“orderer”标志，则命令将尝试在发出升级命令之前，从 Peer 节点中检索信道的 orderer 信息。

下面是 Peer 节点中智能合约的升级命令的一个例子，它将名为 mychannel 的 Channel 中的名为 mycc 的智能合约，从 1.0 版本升级到版本 1.1，其中包含一个新的变量 c。

使用 TLS 和 cafile 全局标志来升级带有 TLS 的网络中的链码：

```
export ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/
```

```
crypto/orderer
    Organizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.exampl
e.com-cert.pem
    peer chaincode upgrade -o orderer.example.com:7050 --tls --cafile $ORDERER_CA -C
mychannel -n mycc -v 1.2 -c '{"Args":["init","a","100","b","200","c","300"]}' -P "OR
('Org1MSP.peer','Org2MSP.peer')"
    .
    .
    .
    2018-02-22 18:26:31.433 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003
Using default escc
    2018-02-22 18:26:31.434 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004
Using default vscc
    2018-02-22 18:26:31.435 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java
chaincode enabled
    2018-02-22 18:26:31.435 UTC [chaincodeCmd] upgrade -> DEBU 006 Get upgrade
proposal for chaincode <name:"mycc" version:"1.1" >
    .
    .
    .
    2018-02-22 18:26:46.687 UTC [chaincodeCmd] upgrade -> DEBU 009 endorse upgrade
proposal, get response <status:200 message:"OK" payload:"\n\004mycc\022\0031.1\032\
004escc\"\004vscc*,\022\014\022\n\010\001\022\002\010\000\022\002\010\001\032\r\022\01
3\n\007Org1MSP\020\003\032\r\022\013\n\007Org2MSP\020\0032f\n
\261g(^v\021\220\240\332\251\014\204V\210P\310o\231\271\036\301\022\032\205fC[|=\215\3
72\223\022 \311b\025?\323N\343\325\032\005\365\236\001XKj\004E\351\007\247\265fu\
305j\367\331\275\253\307R\032 \014H#\014\272!#\345\306s\323\371\350\364\006.\000\356\
230\353\270\263\215\217\303\256\220i^\277\305\214: \375\200zY\275\203}\375\244\205\035
\340\226]l!uE\334\273\214\214\020\303\3474\360\014\234-\006\315B\031\022\010\022\006\
010\001\022\002\010\000\032\r\022\013\n\007Org1MSP\020\001" >
    .
    .
    .
    2018-02-22 18:26:46.693 UTC [chaincodeCmd] upgrade -> DEBU 00c Get Signed envelope
2018-02-22 18:26:46.693 UTC [chaincodeCmd] chaincodeUpgrade -> DEBU 00d Send signed
envelope to orderer
    2018-02-22 18:26:46.908 UTC [main] main -> INFO 00e Exiting.....
```

不使用 TLS 和 cafile 全局标志来升级带有 TLS 的网络中的链码：

```
    peer chaincode upgrade -o orderer.example.com:7050 -C mychannel -n mycc -v 1.2 -c
    '{"Args":["init","a","100","b","200","c","300"]}' -P "OR   ('Org1MSP.peer
','Org2MSP.peer')"
    .
    2018-02-22 18:28:31.433 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003
```

```
Using default escc
    2018-02-22 18:28:31.434 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004
Using default vscc
    2018-02-22 18:28:31.435 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 005 java
chaincode enabled
    2018-02-22 18:28:31.435 UTC [chaincodeCmd] upgrade -> DEBU 006 Get upgrade
proposal for chaincode <name:"mycc" version:"1.1" >
    .
    .
    .
    2018-02-22 18:28:46.687 UTC [chaincodeCmd] upgrade -> DEBU 009 endorse upgrade
proposal, get response <status:200 message:"OK" payload:"\n\004mycc\022\0031.1\032\
004escc\"\004vscc*,\022\014\022\n\010\001\022\002\010\000\022\002\010\001\032\r\022\01
3\n\007Org1MSP\020\003\032\r\022\013\n\007Org2MSP\020\0032f\n \261g(^v\021\220\240\
332\251\014\204V\210P\310o\231\271\036\301\022\032\205fC[|=\215\372\223\022 \311b\025?
\323N\343\325\032\005\365\236\001XKj\004E\351\007\247\265fu\305j\367\331\275\253\307R\
032 \014H#\014\272!#\345\306s\323\371\350\364\006.\000\356\230\353\270\263\215\217\
303\256\220i^\277\305\214: \375\200zY\275\203}\375\244\205\035\340\226]l!uE\334\273\
214\214\020\303\3474\360\014\234-\006\315B\031\022\010\022\006\010\001\022\002\010\
000\032\r\022\013\n\007Org1MSP\020\001" >
    .
    .
    .
    2018-02-22 18:28:46.693 UTC [chaincodeCmd] upgrade -> DEBU 00c Get Signed envelope
2018-02-22 18:28:46.693 UTC [chaincodeCmd] chaincodeUpgrade -> DEBU 00d Send signed
envelope to orderer
    2018-02-22 18:28:46.908 UTC [main] main -> INFO 00e Exiting.....
```

## 7.9 本章小结

本章主要概述了智能合约方面的应用，分别从开发人员以及实际使用人员的视角来讨论智能合约在实际生产过程中的应用。在绝大多数情况下，智能合约的开发人员和使用人员应该是同一角色，会更加深入理解合约的意义与重要性。

在官方提供的操作案例和手册中，智能合约的整体操作过程的复杂程度并非书中所述，更多的需要自我判断具体的使用方案，如打包及未实现功能等。

通过本章的学习，也可以尝试使用更加符合生产需求的方案，将智能合约完全交由服务器端完成，或是通过应用层执行合约上传、命令安装和实例化等操作，在第 9 章讲到的 SDK 可仅用于交易的传递和反馈。

# 第 8 章 CouchDB

## 8.1 CouchDB介绍

HyperLedger Fabric 官方对于 CouchDB 的介绍内容并不多，主要是在富查询阶段推荐使用了该数据库插件。然而，一般建立于区块链平台的项目大多数都会涉及富查询，因此 CouchDB 也成了必不可少的选择。且大多数关于查询的 Chaincode API 都依赖于 CouchDB 实现，就更需要在 Peer 节点所在服务器上搭建一个 CouchDB 服务。

官方的状态数据库包括 LevelDB 和 CouchDB 两种。

LevelDB 是嵌入在 Peer 进程中的默认键/值状态数据库，CouchDB 是一个可选的外部状态数据库。

与 LevelDB 键-值存储一样，CouchDB 可以存储任何以 Chaincode 建模的二进制数据（CouchDB 附件函数在内部用于非 json 二进制数据）。但是，当 Chaincode 值（例如，资产）被建模为 JSON 数据时，则作为 JSON 文档存储，

CouchDB 支持对 Chaincode 数据进行丰富的查询。

LevelDB 和 CouchDB 都支持核心 Chaincode 操作，例如获取和设置一个键（资产），并根据键进行查询。键可以通过范围查询，可以对组合键进行建模，以支持针对多个参数的等价查询。例如，作为所有者的组合键，资产 id 可以用于查询某个实体拥有的所有资产。这些基于 key 的查询可以用于针对账本的只读查询，以及更新总账的事务。

如果将资产建模为 JSON 并使用 CouchDB，就可以使用 Chaincode 中的 CouchDB JSON 查询语句对 Chaincode 数据值执行复杂的富查询，这些类型的查询对于理解账本上的内容很有帮助。

对于这些类型的查询，事务协议响应通常对客户端应用程序有用，但通常不会作为事务提交到排序服务。事实上，也无法保证结果集在 Chaincode 执行与富查询提交时间之间的稳定性，因此使用富查询的结果去执行最终的事务更新操作是不合适的，除非可以保证结果集在 Chaincode 执行时间与提交时间之间的稳定性，或者可以处理在后续交易中的潜在变化。

例如，如果对 Alice 所拥有的所有资产执行一个富查询并将其传输给 Bob，那么一个新的

资产可能会被另一个事务分配给 Alice，这是在 Chaincode 执行时间和提交时间之间的另一个事务，可能此过程中会错过这个“虚值”。

CouchDB 作为一个独立的数据库进程与 Peer 一起运行，因此在设置、管理和操作方面有额外的考虑。可以考虑从默认的嵌入式 LevelDB 开始，如果需要额外的、复杂的富查询，可以转移到 CouchDB。将 Chaincode 资产数据建模为 JSON 是一种很好的做法，这样就可以在将来执行需要的、复杂的富查询。

大多数 Chaincode 的 API 都可以使用 LevelDB 或 CouchDB 状态数据库，例如 GetState、PutState、GetStateByRange 以及 GetStateByPartialCompositeKey 等。

只有当使用 CouchDB 作为状态数据库和在 Chaincode 中作为 JSON 的模型资产时，可以使用 GetQueryResult API 和传递一个 CouchDB 查询字符串来对状态数据库中的 JSON 执行富查询。

---

**注意：**查询字符串遵循 CouchDB JSON 查询语法。具体可以参考 CouchDB 官方网址 http://docs.couchdb.org/en/2.1.1/api/database/find.html。

---

Fabric 项目中的“marbles02”示例演示了使用来自 Chaincode 的 CouchDB 查询。

---

**注意：**marbles02 在 GitHub 上的参考地址为 https://github.com/hyperledger/fabric-samples/blob/master/chaincode/marbles02/go/marbles_chaincode.go。

---

“marbles02”中包括一个 getMarblesByOwner()函数，它通过将一个“owner id”传递给 Chaincode 来演示参数化查询，然后查询与“docType”值为“marble”匹配的 JSON 文档的状态数据，以及“owner id”使用 JSON 查询语法：

```
{
"selector":{
        "docType":"marble",
        "owner":<OWNER_ID>
    }
}
```

为了使 JSON 查询更高效，并且对于任何具有排序的 JSON 查询，都需要使用 CouchDB 中的索引。索引可以和 Chaincode 一起打包 /META-INF/statedb/couchdb/indexes 目录。每个索引必须在其自己的文本文件中通过扩展*.json 被定义，其中索引定义为 JSON 格式并引用 CouchDB 索引 JSON 语法。例如，为了支持上面的 marble 查询，提供了一个关于 docType 和所有者字段的示例索引：

```
{
    "index":{
        "fields":[
            "docType",
            "owner"
        ]
    },
    "ddoc":"indexOwnerDoc",
    "name":"indexOwner",
    "type":"json"
}
```

**注意：** 在 Github 地址为 https://github.com/hyperledger/fabric-samples/blob/master/chaincode/marbles02/go/META-INF/statedb/couchdb/indexes/indexOwner.json 中可以找到示例索引。

**注意：** 在 1.1alpha 版本中，必须为索引定义中引用的每个字段指定“data”封装器。在随后的版本中，指定“data”封装器的需求已经被取消。

在 Chaincode 的 META-INF/statedb/couchdb/indexes 目录中的任何索引，都将与 Chaincode 一起被打包并安装在 Peer 节点上。当 Chaincode 安装在 Peer 节点上，且又在 Peer 节点的 Channel 上实例化时，该索引将自动部署到 Peer 节点的 Channel 状态数据库（如果已配置为使用 CouchDB）。

如果先安装了 Chaincode，然后才在 Channel 上实例化该 Chaincode，那么该索引将会在 Chaincode 实例化的时候进行部署。如果 Chaincode 已经在一个 Channel 上实例化了，然后又在一个 Peer 节点上安装了这个 Chaincode，该索引还是会在 Chaincode 实例化的时候进行部署。

在部署时，索引将自动被 Chaincode 查询利用。CouchDB 可以根据查询中使用的字段自动确定要使用的索引。或者在选择器查询中，可以使用“use_index”关键字指定索引。

在安装的 Chaincode 的后续版本中，可能存在相同的索引。要更改索引，使用相同的索引名称，但要更改索引定义。在安装或实例化 Chaincode 时，索引定义会重新部署到 Peer 节点的状态数据库。

如果所在 Channle 已经拥有大量的数据，并且随后安装了 Chaincode，那么索引的创建在实例化上可能需要一些时间。

类似地，如果 Channel 已经拥有大量的数据并且实例化了后续版本的 Chaincode，那么创

建索引也可能需要一些时间。

所以需要尽量避免在这些时间段内调用查询状态数据库的 Chaincode 函数，因为当索引被初始化时，Chaincode 查询可能会超时。

在事务处理期间，因为区块被提交给账本，所以索引将自动刷新。

通过将状态数据库配置选项从 LevelDB 更改为 CouchDB，可以将 CouchDB 作为状态数据库启用。此外，couchDBAddress 需要配置为指向由 Peer 节点使用的 CouchDB。如果将 CouchDB 配置一个用户名和密码，那么用户名和密码属性应该包含管理员用户名和密码。在 couchDBConfig 中提供了更多的选项，并在适当的地方进行了记录。在重启 Peer 节点之后，修改过的 core.yaml 将会立即生效。

当然，还可以通过 Docker 环境变量来覆盖 core.yaml 值，例如 CORE_LEDGER_STATE_STATEDATABASE 和 CORE_LEDGER_STATE_COUCHDBCONFIG_ COUCHDBADDRESS。

以下是来自 core.yaml 的状态数据库选项：

```
state:
  # stateDatabase - options are "goleveldb", "CouchDB"
  # goleveldb - default state database stored in goleveldb.
  # CouchDB - store state database in CouchDB
  stateDatabase: goleveldb
  couchDBConfig:
     # It is recommended to run CouchDB on the same server as the peer, and
     # not map the CouchDB container port to a server port in docker-compose.
     # Otherwise proper security must be provided on the connection between
     # CouchDB client (on the peer) and server.
     couchDBAddress: couchdb:5984
     # This username must have read and write authority on CouchDB
     username:
     # The password is recommended to pass as an environment variable
     # during start up (e.g. LEDGER_COUCHDBCONFIG_PASSWORD).
     # If it is stored here, the file must be access control protected
     # to prevent unintended users from discovering the password.
     password:
     # Number of retries for CouchDB errors
     maxRetries: 3
     # Number of retries for CouchDB errors during peer startup
     maxRetriesOnStartup: 10
     # CouchDB request timeout (unit: duration, e.g. 20s)
     requestTimeout: 35s
     # Limit on the number of records to return per query
     queryLimit: 10000
```

CouchDB 被托管在 Docker 容器中，通过使用 Docker Compose 脚本利用 Hyperledger

Fabric 设置 CouchDB 的 username 和 password 的能力，通过已有的环境变量设置，即 COUCHDB_USER 和 COUCHDB_PASSWORD 环境变量。

对于在使用 Fabric 提供的 Docker 镜像之外的 CouchDB 安装、本地安装，必须编辑 ini 文件以设置 admin 用户名和密码。

Docker Compose 脚本只在创建容器时设置用户名和密码。如果要在容器创建后更改用户名或密码，则必须对本地.ini 文件进行编辑。

---

**注意：** *在每个 peer 启动时都读取 CouchDB 的 peer 选项。*

---

## 8.2　启动部署

本节主要介绍 CouchDB 如何通过 Peer.yaml 文件配置实现启动。主要可以回头参考第 4.3 节关于 peer.yaml 的编写，其中包含了 CouchDB 和 Peer 节点的配置内容，如下所示：

```
   services:

     couchdb:
       container_name: couchdb
       image: hyperledger/fabric-couchdb
       environment:
       # Comment/Uncomment the port mapping if you want to hide/expose the CouchDB
service,
       # for example map it to utilize Fauxton User Interface in dev environments.
       ports:
         - "5984:5984"

     foo27.org2.example.com:
       container_name: foo27.org2.example.com
       image: hyperledger/fabric-peer
       environment:
         - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
         - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb:5984

         - CORE_PEER_ID=foo27.org2.example.com
         - CORE_PEER_NETWORKID=aberic
         - CORE_PEER_ADDRESS=foo27.org2.example.com:7051
         - CORE_PEER_CHAINCODEADDRESS=foo27.org2.example.com:7052
         - CORE_PEER_CHAINCODELISTENADDRESS=foo27.org2.example.com:7052
         - CORE_PEER_GOSSIP_EXTERNALENDPOINT=foo27.org2.example.com:7051
         - CORE_PEER_LOCALMSPID=Org2MSP
```

```
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      # the following setting starts chaincode containers on the same
      # bridge network as the peers
      # https://docs.docker.com/compose/networking/
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
      - CORE_VM_DOCKER_TLS_ENABLED=false
      # - CORE_LOGGING_LEVEL=ERROR
      - CORE_LOGGING_LEVEL=DEBUG
      - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
      - CORE_PEER_GOSSIP_USELEADERELECTION=true
      - CORE_PEER_GOSSIP_ORGLEADER=false
      - CORE_PEER_PROFILE_ENABLED=false
      - CORE_PEER_TLS_ENABLED=false
      - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
      - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
      - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
    volumes:
        - /var/run/:/host/var/run/
        - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/chaincode/go
        - ./crypto-config/peerOrganizations/org2.example.com/peers/foo27.org2.
example.com/msp:/etc/hyperledger/fabric/msp
        - ./crypto-config/peerOrganizations/org2.example.com/peers/foo27.org2.
example.com/tls:/etc/hyperledger/fabric/tls
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
    command: peer node start
    ports:
      - 7051:7051
      - 7052:7052
      - 7053:7053
    depends_on:
      - couchdb
    networks:
      default:
        aliases:
          - aberic
```

可以看出，其中有一个独立的 CouchDB 启动容器部分，另外在 Peer 节点容器配置中有两个额外的环境变量配置，如下所示：

- CORE_LEDGER_STATE_STATEDATABASE=CouchDB
- CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb:5984

其中，CORE_LEDGER_STATE_STATEDATABASE 的值有两个，分别是“goleveldb”和“CouchDB”，如果不打算启用 CouchDB，这个变量可以不做配置。

Goleveldb 是指存储在 LevelDB 中的默认状态数据库。

CouchDB 是指 CouchDB 中的状态数据库。

建议 CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS 变量在与 Peer 节点相同的服务器上运行 CouchDB，而不是将 CouchDB 容器端口映射到 Docker-Compose 中的服务器端口。否则，必须在 CouchDB 客户端（Peer 节点中的）和服务器之间的连接上提供适当的安全性。其默认值为 127.0.0.1:5984。

当 CouchDB 被成功启动后，可以通过 IP:PORT/_utils 的方式直接在浏览器中对该组件进行访问，结果如图 8-1 所示。

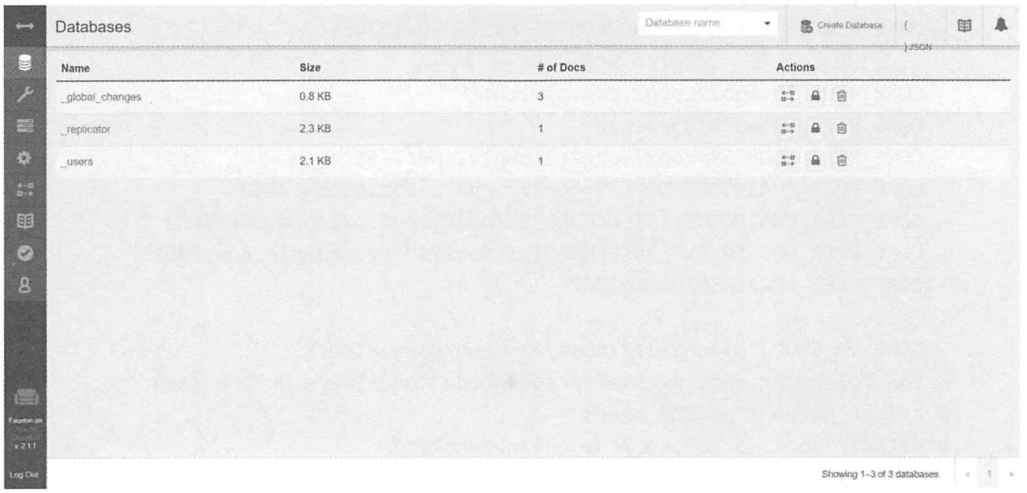


图8–1　登录状态的CouchDB

图 8-1 是按照本节开篇的配置方案启动后进入的界面，但实际生产情况下，应该更加注重访问的安全性和可控性，尤其是有关数据层面的把控。所以，官方在 core.yaml 中的 ledger.state.couchDBConfig 中提供了 username 和 password 两个环境变量。

根据上述，现在对之前的配置文件节选做一些变化，源码结果如下所示：

```
services:

  couchdb:
    container_name: couchdb
    image: hyperledger/fabric-couchdb
    environment:
      - COUCHDB_USER=admin
      - COUCHDB_PASSWORD=123456
    # Comment/Uncomment the port mapping if you want to hide/expose the CouchDB
```

```
service,
       # for example map it to utilize Fauxton User Interface in dev environments.
       ports:
         - "5984:5984"

     foo27.org2.example.com:
       container_name: foo27.org2.example.com
       image: hyperledger/fabric-peer
       environment:
         - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
         - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb:5984
         - CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=admin
         - CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=123456

         - CORE_PEER_ID=foo27.org2.example.com
         - CORE_PEER_NETWORKID=aberic
         - CORE_PEER_ADDRESS=foo27.org2.example.com:7051
         - CORE_PEER_CHAINCODEADDRESS=foo27.org2.example.com:7052
         - CORE_PEER_CHAINCODELISTENADDRESS=foo27.org2.example.com:7052
         - CORE_PEER_GOSSIP_EXTERNALENDPOINT=foo27.org2.example.com:7051
         - CORE_PEER_LOCALMSPID=Org2MSP

         - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
         # the following setting starts chaincode containers on the same
         # bridge network as the peers
         # https://docs.docker.com/compose/networking/
         - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=aberic_default
         - CORE_VM_DOCKER_TLS_ENABLED=false
         # - CORE_LOGGING_LEVEL=ERROR
         - CORE_LOGGING_LEVEL=DEBUG
         - CORE_PEER_GOSSIP_SKIPHANDSHAKE=true
         - CORE_PEER_GOSSIP_USELEADERELECTION=true
         - CORE_PEER_GOSSIP_ORGLEADER=false
         - CORE_PEER_PROFILE_ENABLED=false
         - CORE_PEER_TLS_ENABLED=false
         - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
         - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
         - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
       volumes:
           - /var/run/:/host/var/run/
           - ./chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/chaincode/go
           - ./crypto-config/peerOrganizations/org2.example.com/peers/foo27.org2.
example.com/msp:/etc/hyperledger/fabric/msp
           - ./crypto-config/peerOrganizations/org2.example.com/peers/foo27. org2.
```

```
example.com/tls:/etc/hyperledger/fabric/tls
      working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      command: peer node start
      ports:
        - 7051:7051
        - 7052:7052
        - 7053:7053
      depends_on:
        - couchdb
      networks:
        default:
          aliases:
            - aberic
```

可以看到，在 foo27.org2.example.com 节点下新增了 CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME 和 CORE_LEDGER_STATE_COUCHDBCONFIG_ PASSWORD 两个环境变量，并且分别赋值为“admin”和“123456”。同时在 CouchDB 节点也新增了 COUCHDB_USER 和 COUCHDB_PASSWORD 两个环境变量，赋值与 Peer 节点下的一致。

在 CouchDB 节点下新增的两个环境变量表示即将启动的 CouchDB 的用户名和密码分别是 admin 及 123456。而对于 Peer 节点下新增的两个环境变量，则意味着当 Peer 节点有需求地执行智能合约，并进入 CouchDB 进行查询操作的时候，所需要用到的用户名和密码。

当再次启动 CouchDB 容器并通过 IP:PORT/_utils 访问其浏览器主界面时，会得到如图 8-2 所示登录页。

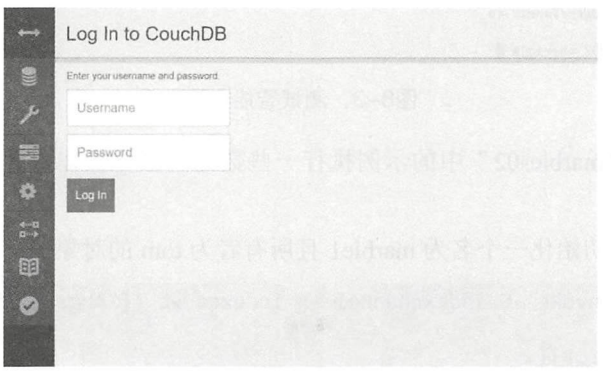


图8-2　CouchDB登录页

此时的 CouchDB，如果是被部署在有外网访问权限的服务器上，也相比之前具有了一定的安全性，且此刻 CouchDB 的所有基于 Restful 风格的 API 都必须持有当前 CouchDB 的用户名和密码才能请求到有效结果。

这样的启动方式将极大地提升服务安全性，为已有数据增加了一重保障。

当在该页面输入用户名 admin 和密码 123456 之后，所出现的界面将与图 8-1 一样。

## 8.3 索引应用

第 8.1 节提到了 CouchDB 的索引，也提到了官方提供的名为“marbles02”的案例，本节就以“marbles02”为案例阐述一下索引的相关问题。

“marbles02”位于 Fabric 源码 fabric\examples\chaincode\go 目录下，本节以第 6 章的环境为基础，以 172.31.143.21 服务器为媒介进行演示。

将该 fabric\examples\chaincode\go 目录下的 marbles02 目录上传至 172.31.143.21 服务器的 /home/docker/github.com/hyperledger/fabric/aberic/chaincode/go 目录下。

进入 172.31.143.21 服务器终端，并根据第 4 章中的方案创建一个明文 indexchannel 的频道，同时在该频道下安装一个名为 indexcc 的智能合约，该智能合约的路径与“marbles02”相同，随后实例化该智能合约，最终得到如图 8-3 所示结果。

```
[root@iZm5edfjk3nwwk6rlnmed9Z aberic]# docker ps
CONTAINER ID        IMAGE                                                
```

```
    Query Result:
{"color":"blue","docType":"marble","name":"marble1","owner":"tom","si
    ze":35}
    2018-04-18 12:49:20.815 UTC [main] main -> INFO 008 Exiting.....
```

说明数据初始化成功。接下来，继续执行如下命令，分别初始化一个名为 marble2 和 marble3 且所有者为 tom 的对象：

```
    peer chaincode invoke -C indexchannel -n indexcc -c '{"Args":["initMarble","
marble2
    ","red","50","tom"]}'
    peer chaincode invoke -C indexchannel -n indexcc -c '{"Args":["initMarble","
marble3","blue","70","tom"]}'
```

此时，Peer 节点中应该已经有三个所有者为 tom 的对象，执行如下命令进行查询：

```
    peer chaincode query -C indexchannel -n indexcc -c '{"Args":["queryMarblesByOwner
","
    tom"]}'
```

可以得到如下日志结果：

```
    Query Result: [{"Key":"marble1", "Record":{"color":"blue","docType":"marble","name
    ":"marble1","owner":"tom","size":35}},{"Key":"marble2", "Record":{"color":"
red","docType":"marble","name":"marble2","owner":"tom","size":50}},{"Key":"marble3",
"Record":{"color":"blue","docType":"marble","name":"marble3","owner":"tom","size":70}}]
    2018-04-18 13:00:45.901 UTC [main] main -> INFO 008 Exiting.....
```

上述日志中表明所有者为 tom 的 marble 对象都已经被查询出来，下面介绍 queryMarblesByOwner 方法，具体源码如下：

```
    func (t *SimpleChaincode) queryMarblesByOwner(stub shim.ChaincodeStubInterface,
args
     []string) pb.Response {

    //   0
    // "bob"
    if len(args) < 1 {
        return shim.Error("Incorrect number of arguments. Expecting 1")
    }

    owner := strings.ToLower(args[0])

    queryString := fmt.Sprintf("{\"selector\":{\"docType\":\"marble\",\"owner\":\
"%s\"}}", owner)
```

```
    queryResults, err := getQueryResultForQueryString(stub, queryString)
    if err != nil {
        return shim.Error(err.Error())
    }
    return shim.Success(queryResults)
}
```

在此方法中，由接入的请求参数数组组成了一个查询语句，该语句符合 CouchDB 的语法。随后，又将拼接好的查询语句传入名为“getQueryResultForQueryString”的方法中。接下来，看下该方法的源码，如下：

```
func getQueryResultForQueryString(stub shim.ChaincodeStubInterface, queryString
stri
ng) ([]byte, error) {

    fmt.Printf("- getQueryResultForQueryString queryString:\n%s\n", queryString)

    resultsIterator, err := stub.GetQueryResult(queryString)
    if err != nil {
        return nil, err
    }
    defer resultsIterator.Close()

    // buffer is a JSON array containing QueryRecords
    var buffer bytes.Buffer
    buffer.WriteString("[")

    bArrayMemberAlreadyWritten := false
    for resultsIterator.HasNext() {
        queryResponse, err := resultsIterator.Next()
        if err != nil {
            return nil, err
        }
        // Add a comma before array members, suppress it for the first array member
        if bArrayMemberAlreadyWritten == true {
            buffer.WriteString(",")
        }
        buffer.WriteString("{\"Key\":")
        buffer.WriteString("\"")
        buffer.WriteString(queryResponse.Key)
        buffer.WriteString("\"")

        buffer.WriteString(", \"Record\":")
        // Record is a JSON object, so we write as-is
```

```
        buffer.WriteString(string(queryResponse.Value))
        buffer.WriteString("}")
        bArrayMemberAlreadyWritten = true
    }
    buffer.WriteString("]")

    fmt.Printf("- getQueryResultForQueryString queryResult:\n%s\n", buffer.String())

    return buffer.Bytes(), nil
}
```

在 getQueryResultForQueryString 方法中，执行了 Chaincode API 中的 GetQueryResult 方法执行了传入的查询语句。通过第 7.7 节可以得知，GetQueryResult 方法是仅支持 CouchDB 的富查询方法，它返回了一个迭代器，随后通过迭代器中的循环操作取出了希望查询到的值。

另外，可以通过 IP:PORT/_utils 再次访问 CouchDB 页面。本节中的访问地址为 172.31.143.21:5984/_utils。

通过上述方案访问后，可以得到如图 8-4 所示结果。

Databases　　Database name　　Create Database

| Name | Size | # of Docs | Actions |
|---|---|---|---|
| _global_changes | 2.8 KB | 9 | |
| _replicator | 2.3 KB | 1 | |
| _users | 2.1 KB | 1 | |
| indexchannel_ | 6.8 KB | 2 | |
| indexchannel_indexcc | 2.7 KB | 7 | |
| indexchannel_lscc | 0.6 KB | 1 | |

图8-4　CouchDB页面

可以看到，之前创建的名为 indexcc 的智能合约以及它所属的名为 indexchannel 的 Channel。在之前的操作中执行了三次初始化操作，理论上应该只有 3 条数据，而根据上图可以看到在 indexchannel_indexcc 竟然有 7 条数据。通过点击，查看到底是哪 7 条数据，如图 8-5 所示。

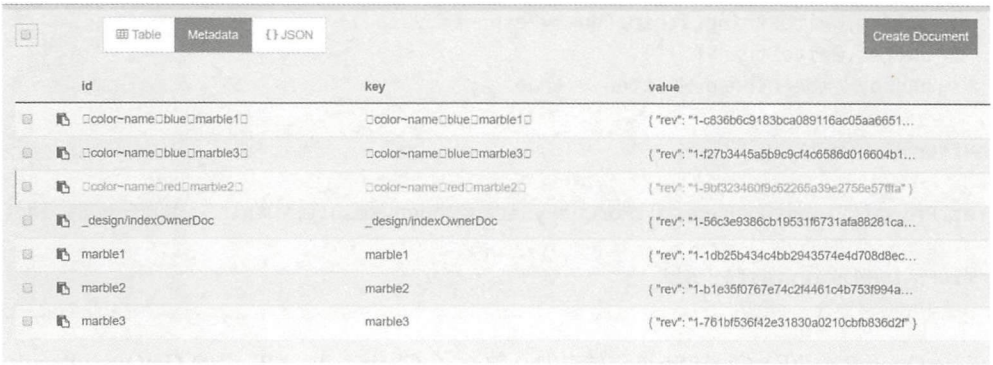


图8-5　CouchDB数据

在结果中发现，有 3 条 id 为 marble*类型的数据，继续分别进入这三条数据查看详情，并以 marble1 为例，如图 8-6 所示。

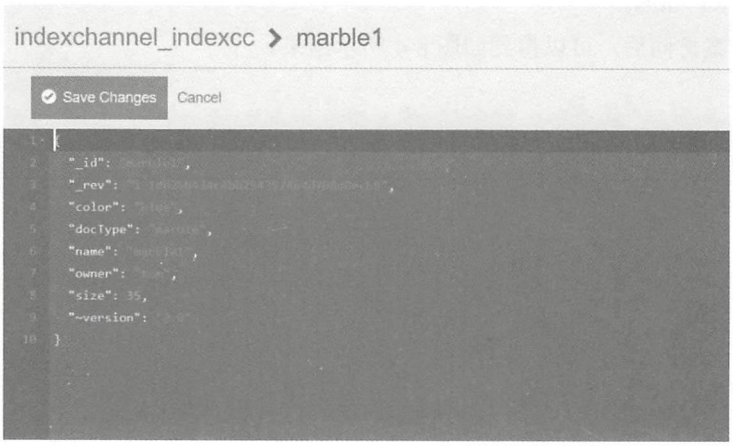


图8-6　marble1数据

在 marble1 中，列出了之前初始化 marble1 时的所有参数，且与初始化的参数一一对应。这条数据也与初始化方法 initMarble 所做一致，具体的初始化方法 initMarble 源码如下：

```
   func (t *SimpleChaincode) initMarble(stub shim.ChaincodeStubInterface, args
[]strin
   g) pb.Response {
    var err error

    //   0      1      2     3
    // "asdf", "blue", "35", "bob"
    if len(args) != 4 {
        return shim.Error("Incorrect number of arguments. Expecting 4")
```

```
    }

    // ==== Input sanitation ====
    fmt.Println("- start init marble")
    if len(args[0]) <= 0 {
        return shim.Error("1st argument must be a non-empty string")
    }
    if len(args[1]) <= 0 {
        return shim.Error("2nd argument must be a non-empty string")
    }
    if len(args[2]) <= 0 {
        return shim.Error("3rd argument must be a non-empty string")
    }
    if len(args[3]) <= 0 {
        return shim.Error("4th argument must be a non-empty string")
    }
    marbleName := args[0]
    color := strings.ToLower(args[1])
    owner := strings.ToLower(args[3])
    size, err := strconv.Atoi(args[2])
    if err != nil {
        return shim.Error("3rd argument must be a numeric string")
    }

    // ==== Check if marble already exists ====
    marbleAsBytes, err := stub.GetState(marbleName)
    if err != nil {
        return shim.Error("Failed to get marble: " + err.Error())
    } else if marbleAsBytes != nil {
        fmt.Println("This marble already exists: " + marbleName)
        return shim.Error("This marble already exists: " + marbleName)
    }

    // ==== Create marble object and marshal to JSON ====
    objectType := "marble"
    marble := &marble{objectType, marbleName, color, size, owner}
    marbleJSONasBytes, err := json.Marshal(marble)
    if err != nil {
        return shim.Error(err.Error())
    }
    //Alternatively, build the marble json string manually if you don't want to use
struct marshalling
    //marbleJSONasString := `{"docType":"Marble",  "name": "` + marbleName + `",
"color": "` + color + `", "size": ` + strconv.Itoa(size) + `, "owner": "` + owner +
```

```
`"}`
    //marbleJSONasBytes := []byte(str)

    // === Save marble to state ===
    err = stub.PutState(marbleName, marbleJSONasBytes)
    if err != nil {
        return shim.Error(err.Error())
    }

    //  ==== Index the marble to enable color-based range queries, e.g. return all
blue marbles ====
    //  An 'index' is a normal key/value entry in state.
    //  The key is a composite key, with the elements that you want to range query on
listed first.
    //  In our case, the composite key is based on indexName~color~name.
    //  This will enable very efficient state range queries based on composite keys
matching indexName~color~*
    indexName := "color~name"
   colorNameIndexKey, err := stub.CreateCompositeKey(indexName, []string{marble.Color,
marble.Name})
    if err != nil {
        return shim.Error(err.Error())
    }
    //  Save index entry to state. Only the key name is needed, no need to store a
duplicate copy of the marble.
    //  Note - passing a 'nil' value will effectively delete the key from state,
therefore we pass null character as value
    value := []byte{0x00}
    stub.PutState(colorNameIndexKey, value)

    // ==== Marble saved and indexed. Return success ====
    fmt.Println("- end init marble")
    return shim.Success(nil)
    }
```

在初始化方法中，将 marblename 作为 key 存入账本中，而 marblename 就是现在的 marble1。

那么，marble*这 3 条数据的来源解决了，理论上也符合了最开始的预期。接下来，继续看另外 4 条数据。从图 8-5 中可以看到 id 为_design/indexOwnerDoc 的数据是唯一一条无明显规律的数据，点进去查看详情，如图 8-7 所示。

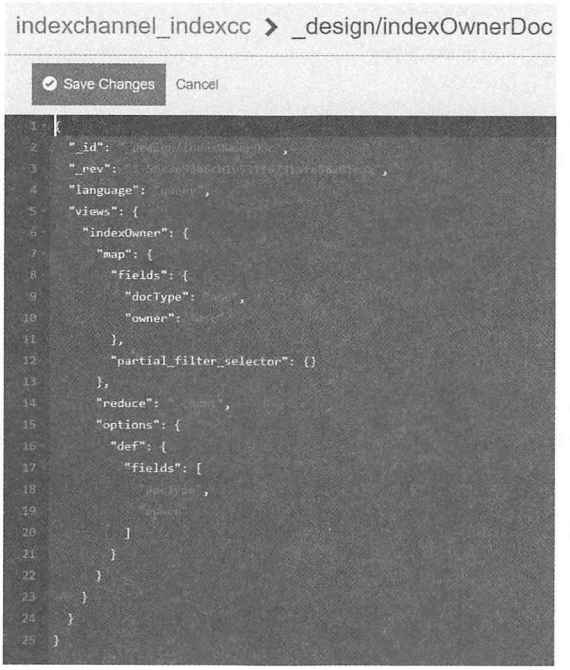


图8-7　index内容

可以看到其中的 index 关键字，这是一条索引。这条索引是什么时候创建的，又怎么创建索引？

通过第 8.1 节中的介绍，索引可以和 Chaincode 一起打包在/META-INF/statedb/couchdb/Indexes 目录下，每个索引必须在其自己的文本文件中通过扩展*.json 被定义，其中索引定义为 JSON 格式并引用 CouchDB 索引 JSON 语法。

在安装名为 indexcc 智能合约时，上传了源码 fabric\examples\chaincode\go 目录下的 marbles02 文件夹，在该文件夹下的/META-INF/statedb/couchdb/indexes 目录下就有一个名为 indexOwner.json 的文件，其中 json 内容如下：

```
{"index":{"fields":["docType","owner"]},"ddoc":"indexOwnerDoc", "name":"indexOwner
","type":"json"}
```

即创建了一个 json 类型且名为 indexOwner 的索引，其索引文档名为 indexOwnerDoc，索引字段分别是 docType 和 owner。

在进入 CouchDB 页 indexchannel_indexcc 库中，可以找到名为 indexOwnerDoc 的文档链接，但在创建索引文档时会统一在之前加上“_design/”。

关于索引字段 docType 和 owner 可以参考 marbles02 中的 struct，源码如下：

```
type marble struct {
 ObjectType string `json:"docType"`
 Name       string `json:"name"`
 Color      string `json:"color"`
 Size       int    `json:"size"`
 Owner      string `json:"owner"`
}
```

在 struct 中分别对 ObjectType 和 Owner 执行了其存档的 json 格式的 key 分别是 docType 和 owner，与索引关联字段对应。

现在回头再看 queryMarblesByOwner 方法中拼接的查询语句，正是使用索引的方式查询 docType 和 owner 字段的相关内容。

再去看 CouchDB 中多出来的剩余 3 条数据，则为该索引规则创建出来的 3 条索引文档。

利用 CouchDB 创建索引可以极大地提高查询效率，尤其是面对大数据量进行富查询的时候。如果没有索引，那么触发超时的概率将会很高。所以，索引也将在 HyperLedger Fabric 平台中广泛引用。

但索引不能滥用，它是一种用空间换时间的方案，所以在智能合约设计之初就应该做好相关的准备，避免创建大量的索引而引发的存储膨胀等一系列问题。

接下来将继续这个案例，讲解如何在已经实例化过的智能合约中创建一个新的索引，而并非与智能合约一同打包进去。

其实，在 marbles02 的源码中，官方已经给出了大量示例来概述这一块的内容，这里仅对官方示例做一次演示。

本例将创建一个以 Name 和 Color 为关联的索引，具体可通过两种方案来创建，第一种是通过 Post 请求，第二种是直接在 CouchDB 页面中创建。

首先看第一种方案，通过 Post 请求方式，对 CouchDB 指定接口执行 Post 请求并发送一个 json 字符串以创建索引。这里直接通过服务器中 curl 命令实现，具体命令如下所示：

```
curl -i -X POST -H "Content-Type: application/json" -d "{\"index\":{\"fields\":
[\"na
me\",\"color \"]},\"name\":\"indexColor\",\"ddoc\":\"indexColorDoc\",\"type\
":\"json\"}" http://admin:123456@172.31.143.21:5984/indexchannel_indexcc/_index
```

---

**注意**：在对 CouchDB 中 restful 接口发送请求的时候，如果已经设置了其用户名和密码，则必须在 host 前加上“username:password@”，以便以合法的登录状态执行接口，否则会出现类似“{"error":"unknown_error","reason":"undef","ref":278200450}”的错误反馈。

---

随后，再进入 indexchannel_indexcc，可以看到多出一条名为“_design/indexColorDoc”的

数据，这就是通过 Post 请求创建的新的索引。

接下来，创建另外一条索引，并直接在 CouchDB 页面中创建。这次为 Name 和 Size 字段创建关联索引，进入 CouchDB 页并参考图 8-8 进入 Mango Indexes。

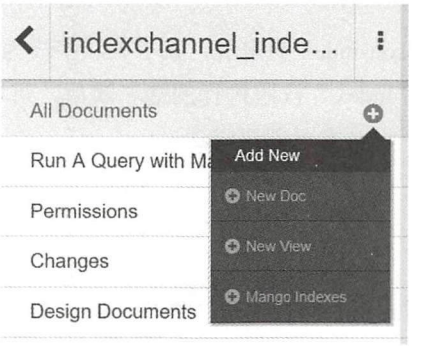


图8-8　创建索引

进入索引集合页面后，可以看到右侧是已经建好的索引列表，而左侧则是新建索引的操作入口，并且已经给出了建立单字段或多字段的案例，如图 8-9 所示。

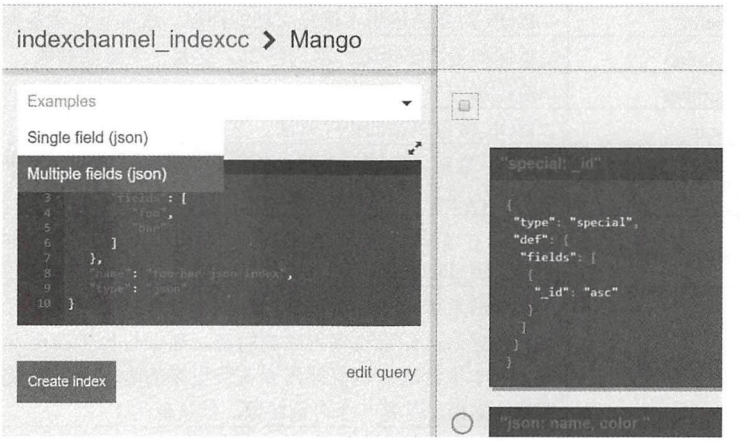


图8-9　新建索引

选择 Multiple fields(json)，按照给出的案例并修改成之前所述的两个字段，最终结果如下：

```
{
 "index": {
     "fields": [
"name",
"size"
]
```

```
  },
  "name": "indexSize",
  "ddoc": "indexSizeDoc",
  "type": "json"
}
```

创建一个名为 name-size-json-index 的索引，索引关联字段为 name 和 size 且为 json 类型，粘贴如输入框后单击“Create Index”按钮执行创建。随后，可以在右侧看到新增了一条明文"json: name, size"的索引文档，说明索引创建成功。

## 8.4 查询应用

这里的查询是指使用声明式 JSON 查询语法查找文档。查询可以使用内置的_all_docs 索引或自定义索引，这些索引使用_index 符指定。

对 CouchDB 执行查询时，需要传入的 Json 参数见表 8-1。

表 8-1 参数集

| 参 数 | 类 型 | 描 述 |
| --- | --- | --- |
| selector | json | JSON 对象描述用于选择文档的标准。这是一个必备参数 |
| limit | number | 返回的最大结果数。默认是 25。这是一个可选参数 |
| skip | number | 跳过第一个“n”结果，其中“n”是指定的值。这是一个可选参数 |
| sort | json | JSON 数组遵循的排序语法。这是一个可选参数 |
| fields | array | JSON 数组，指定每个对象的哪些字段应该返回。如果省略了，就返回整个对象。这是一个可选参数 |
| use_index | string\|array | 指示一个查询使用一个特定的索引。指定为"<design_document>"或["<design_document>", "<index_name>"]。这是一个可选参数 |
| r | number | 读取结果所需的文档数量。默认为 1，在这种情况下，在索引中找到的文档会返回。如果设置为更高的值，那么每份文档至少在返回结果之前的许多副本中读取。这可能需要花费更多的时间，而不仅仅是使用索引本地存储的文档。这是一个可选参数。默认值为 1 |
| bookmark | string | 这是一个字符串，它使用户能够指定所需要的结果页面。用于通过结果集进行分页。每个查询都会在书签 key 下返回一个不透明的字符串，然后在查询中返回，以获得下一页的结果。如果选择器查询的任何部分在请求之间发生变化，那么结果是未定义的。这是一个可选参数 |
| update | boolean | 是否在返回结果之前更新索引。默认是 true。这是一个可选参数 |
| stable | boolean | 视图的结果是否应该从“稳定版”一组集合中返回。这是一个可选参数 |
| stale | string | update=false 和 stable=true 选项。可能赋值:“ok”。这是一个可选参数 |
| execution_stats | boolean | 在查询回复中包含执行统计信息，默认为 false。这是一个可选参数 |

（1）对 CouchDB 执行查询后，返回的 Json 参数见表 8-2。

表 8-2　返回参数集

| 参　数 | 类　型 | 描　述 |
| --- | --- | --- |
| docs | object | 匹配搜索的文档数组。在每一个匹配的文档中，列出了请求正文部分中指定的字段，以及它们的值 |
| warning | string | 警告内容 |
| execution_stats | object | 统计数据对象 |

（2）对 CouchDB 执行查询后的返回状态见表 8-3。

表 8-3　返回状态集

| 参　数 | 类　型 | 描　述 |
| --- | --- | --- |
| 200 | OK | 请求成功完成 |
| 400 | Bad Request | 无效的请求 |
| 401 | Unauthorized | 读权限请求 |
| 500 | Internal Server Error | 查询执行错误 |

**注意**：limit 和 skip 的结果与一般数据库的用法和返回值完全一致。虽然 skip 存在，但它并不打算用于分页。原因是 bookmark 功能更高效。

请求参数示例，示例请求主体使用索引查找文档的 Json 对象：

```
{
   "selector": {
      "year": {"$gt": 2010}
   },
   "fields": ["_id", "_rev", "year", "title"],
   "sort": [{"year": "asc"}],
   "limit": 2,
   "skip": 0,
   "execution_stats": true
}
```

请求响应示例，在使用索引查找文档时的示例响应的 Json 对象：

```
{
   "docs": [
      {
         "_id": "176694",
         "_rev": "1-54f8e950cc338d2385d9b0cda2fd918e",
         "year": 2011,
```

```
            "title": "The Tragedy of Man"
        },
        {
            "_id": "780504",
            "_rev": "1-5f14bab1a1e9ac3ebdf85905f47fb084",
            "year": 2011,
            "title": "Drive"
        }
    ],
    "execution_stats": {
        "total_keys_examined": 0,
        "total_docs_examined": 200,
        "total_quorum_docs_examined": 0,
        "results_returned": 2,
        "execution_time_ms": 5.52
    }
}
```

## 8.5 选择器语法

选择器（Selector）表示为一个期望正确获取的文档的 JSON 对象。在这个结构中，可以使用专门命名的字段来应用查询的条件逻辑。

虽然选择器与 MongoDB 查询文档有一些相似之处，但它们来自于目标的相似性，并不一定扩展到功能或结果的共性。

### 8.5.1 基本语法

最基础的选择器语法要求指定一个或多个字段，以及这些字段所需的相应值。即编写一个简单的选择器，检查特定的字段。例如：这个选择器匹配所有“director”字段中具有“Lars von Trier”值的文档。如下所示本段示例：

```
{
    "director": "Lars von Trier"
}
"selector": {
  "$text": "Bond"
},
"fields": [
  "title",
  "cast"
]
```

可以通过组合运算符来创建更复杂的选择器表达式。为了获得最佳性能，最好将“组合”或“数组逻辑”运算符组合在一起，比如$regex，以及$eq、$gt、$gte、$lt 和$lte（但没有$ne）这样的平等运算符。

下面是一个选择器查询关联两个字段的示例。

这个选择器匹配任何带有“Paul”的名称字段的文档，并且它也有一个值“Boston”的位置字段。

```
{
   "name": "Paul",
   "location": "Boston"
}
```

## 8.5.2　嵌套对象

更复杂的选择器可以指定嵌套对象或子字段的值。例如，可以使用标准的 JSON 结构来指定字段和子字段。

字段和子字段选择器的示例，使用标准的 JSON 结构：

```
{
   "imdb": {
      "rating": 8
   }
}
```

可以使用缩写的方式，即使用点符号将字段和子字段名合并成一个单独的名称，也能够起到等价作用，如下示例：

```
{
   "imdb.rating": 8
}
```

## 8.5.3　运算符

运算符通过在 name 字段中使用美元符号（$）前缀来标志。

在选择器语法中，有两种核心类型的运算符，分别是组合运算符和条件运算符。

一般来说，组合运算符被应用于最顶层的选择。它们被用来组合条件，或者将条件组合成一个选择器。

每个显式运算符都有如下形式：

```
{"$operator": argument}
```

没有显式运算符的选择器被认为具有隐式选择器。确切的隐式运算符是由选择器表达式的

结构决定的。

### 8.5.4 隐式运算符

有两个隐式运算符，分别是“平等（Equality）”和“与（And）”。

在选择器中，任何含有 JSON 值的字段，但其中没有运算符，都被认为是一种平等条件。隐式的平等测试也适用于字段和子字段。

任何不是对条件运算符的参数的 JSON 对象都是每个字段的隐式$运算符。

在下面的例子中，使用一个运算符来匹配任何文档，其中 year 字段的值大于 2010。示例如下：

```
{
   "year": {
      "$gt": 2010
   }
}
```

在下一个例子中，必须在匹配的文档中有一个字段“director”，并且该字段必须有一个完全等于“Lars von Trier”的值，示例如下：

```
{
   "director": "Lars von Trier"
}
```

也可以使上述表达式写成一个显示运算符的形式，示例如下：

```
{
   "director": {
      "$eq": "Lars von Trier"
   }
}
```

在下一个使用子字段的例子中，匹配文档中必需的字段“imdb”，也必须有一个子字段“rating”，子字段的值必须等于 8。示例如下：

```
{
   "imdb": {
      "rating": 8
   }
}
```

当然，也可以把上述表达式改成显示形式，示例如下：

```
{
```

```
    "imdb": {
       "rating": { "$eq": 8 }
    }
}
```

下面是一个使用全文索引的$eq 运算符的例子：

```
{
  "selector": {
    "year": {
      "$eq": 2001
    }
  },
  "sort": [
    "title:string"
  ],
  "fields": [
    "title"
  ]
}
```

下面是在字段“year”索引中使用的$eq 运算符的一个例子：

```
{
  "selector": {
    "year": {
      "$eq": 2001
    }
  },
  "sort": [
    "year"
  ],
  "fields": [
    "year"
  ]
}
```

在本例中，必须存在字段“director”，并包含“Lars von Trier”的值，并且字段“year”必须存在，且它的值为 2003。

```
{
   "director": "Lars von Trier",
   "year": 2003
}
```

上述可以使$and 运算符和等式运算符显式。使用显式$和$eq 运算符如下所示：

```
{
    "$and": [
        {
            "director": {
                "$eq": "Lars von Trier"
            }
        },
        {
            "year": {
                "$eq": 2003
            }
        }
    ]
}
```

### 8.5.5 显示运算符

除了“Equality”和“And”之外，所有的运算符都必须显示声明。

1. 组合运算符

组合运算符用于组合选择器。除了在大多数编程语言中出现的普通布尔运算符之外，还有三个组合运算符（$all、$elemMatch 和$allMatch）帮助处理 JSON 数组。

组合运算符只需要一个参数。这个参数要么是另一个选择器，要么是一个选择器数组。

组合运算符参数见表 8-4。

表 8-4 组合运算符参数

| 运算符 | 参数类型 | 备 注 |
| --- | --- | --- |
| $and | Array | 数组中的所有选择器匹配，则匹配 |
| $or | Array | 数组中的任何选择器匹配，则匹配。所有选择器必须使用相同的索引 |
| $not | Selector | 给定的选择器不匹配的话 |
| $nor | Array | 数组中的选择器没有匹配的话 |
| $all | Array | 包含参数数组的所有元素，则匹配数组值 |
| $elemMatch | Selector | 匹配并返回包含数组字段的所有文档，其中至少有一个与所有指定查询条件匹配的元素 |
| $allMatch | Selector | 匹配并返回包含数组字段的所有文档，其中所有元素都匹配所有指定的查询条件 |

（1）$and 运算符。下面是使用全文索引的$and 运算符示例：

```
{
  "selector": {
    "$and": [
```

```
      {
        "$text": "Schwarzenegger"
      },
      {
        "year": {
          "$in": [1984, 1991]
        }
      }
    ]
  },
  "fields": [
    "year",
    "title",
    "cast"
  ]
}
```

如果 Json 中的所有选择器匹配，则$and 运算符匹配。下面是一个使用主索引（“_all_docs”）的示例：

```
{
   "$and": [
      {
         "_id": { "$gt": null }
      },
      {
         "year": {
            "$in": [2014, 2015]
         }
      }
   ]
}
```

（2）$or 运算符。如果 Json 中的任何一个选择器匹配，则$or 运算符匹配。下面是一个在“year”上使用索引的例子：

```
{
   "year": 1977,
   "$or": [
      { "director": "George Lucas" },
      { "director": "Steven Spielberg" }
   ]
}
```

（3）$not 运算符。如果给定的选择器不匹配，则$not 运算符匹配。下面是一个在“year”

上使用索引的例子：

```
{
    "year": {
        "$gte": 1900
    },
    "year": {
        "$lte": 1903
    },
    "$not": {
        "year": 1901
    }
}
```

（4）$nor 运算符。如果给定的选择器不匹配，则$nor 运算符匹配。下面是一个在“year”上使用索引的例子：

```
{
    "year": {
        "$gte": 1900
    },
    "year": {
        "$lte": 1910
    },
    "$nor": [
        { "year": 1901 },
        { "year": 1905 },
        {  "year": 1907 }
    ]
}
```

（5）$all 运算符。如果包含参数数组的所有元素，则$all 运算符匹配数组值。下面是一个用于主索引（_all_docs）的示例：

```
{
    "_id": {
        "$gt": null
    },
    "genre": {
        "$all": ["Comedy","Short"]
    }
}
```

（6）$elemMatch 运算符。$elemMatch 运算符匹配并返回包含数组字段的所有文档，其中至少有一个与提供的查询条件匹配的元素。下面是一个用于主索引（_all_docs）的示例：

```
{
    "_id": { "$gt": null },
    "genre": {
        "$elemMatch": {
            "$eq": "Horror"
        }
    }
}
```

（7）$allMatch 运算符。$allMatch 运算符匹配并返回包含数组字段的所有文档，其中包含匹配提供的查询条件的所有元素。下面是一个用于主索引（_all_docs）的示例：

```
{
    "_id": { "$gt": null },
    "genre": {
        "$allMatch": {
            "$eq": "Horror"
        }
    }
}
```

2. 条件运算符

条件运算符是特定于字段的，用于评估存储在该字段中的值。例如，当指定的字段包含一个等于提供的参数的值时，基本的$eq 运算符匹配。

大多数编程语言都支持基本的平等和不平等运算符。此外，还可以使用一些“meta”条件运算符。条件运算符接受任何有效的 JSON 内容作为参数。其他条件运算符要求参数为特定的 JSON 格式。

条件运算符参数见表 8-5。

表 8-5　条件运算符参数

| 运算符类型 | 运算符 | 参　数 |
| --- | --- | --- |
| (In)equality/比较 | $lt | Any JSON |
| | $lte | Any JSON |
| | $eq | Any JSON |
| | $ne | Any JSON |
| | $gte | Any JSON |
| | $gt | Any JSON |
| Object/对象 | $exists | Boolean |
| | $type | String |

续表

| 运算符类型 | 运算符 | 参　数 |
| --- | --- | --- |
| Array/数组 | $in | Array of JSON values |
| | $nin | Array of JSON values |
| | $size | Integer |
| Miscellaneous | $mod | [Divisor, Remainder] |
| | $regex | String |

## 8.6 本章小结

本章主要概述了 CouchDB 的基本用法以及相对较为深入一些的索引操作。从某种程度上来说，CouchDB 是在没有推出新型的数据库插件（如：BlockDB 等）之前必须用来替代 LevelDB 的解决方案。所以，关于 CouchDB 在索引方面的应用必须实际测试并彻底掌握，在编写智能合约之前，需要对合约中实体的结构更加细致地分析和使用，尤其是与 CouchDB 结合的情况，便于通过索引实现一次性获取以达到目的的效果。

# 第 9 章　Java-SDK 客户端

HyperLedger Fabric 计划为各种各样的编程语言提供大量的 SDK。首先提供了 Node.js 和 Java SDK。在随后的版本中，将陆续提供可生产使用的 Python、Restful 和 Go SDK。

尤其是在 HyperLedger Fabric1.0 版本之后，官方开始极力推荐使用 SDK 来实现交互的操作，原本在 0.6 版本上的 Restful API 已经被废弃。JAVA-SDK 可以参考 GitHub 地址为 https://github.com/hyperledger/fabric-sdk-java，Node-SDK 可以参考 GitHub 地址为 https://github.com/hyperledger/fabric-sdk-node。

本书的 SDK 是以 Fabric-Java-SDK 为基础，该 SDK 有助于促进 Java 应用程序管理 HyperLedger Channel 和用户智能合约的生命周期。SDK 还提供了一种方法来执行用户智能合约、查询区块和 Channel 上的交易，并监控 Channel 上的事件。

SDK 没有为应用程序定义的 Channel 和客户端构件提供持久性的方法。这将留给嵌入应用程序以最佳管理。Channel 可以在客户端的上下文中通过 Java 序列化方法进行序列化。反序列化的 Channel 不处于初始状态。应用程序需要处理不同版本之间的事务和事务状态管理。

该 SDK 还为 HyperLedger 的认证机构提供了一个客户端。然而，SDK 并不依赖于证书颁发机构的特定实现。其他的证书颁发机构可以通过实现 SDK 的注册接口来使用。

官方提供的 SDK 的 DEMO 较难入手，本书在官方 SDK DEMO 的基础上做了一次整理，能够更加清晰简单地入门并使用 JAVA-SDK 进行交互。

## 9.1　SDK项目前置条件

编写并使用 Fabric-Java-SDK 时，可以使用 eclipse 或者 IDEA 作为编辑器。本书以 eclipse 为例，大致目录结构如图 9-1 所示。

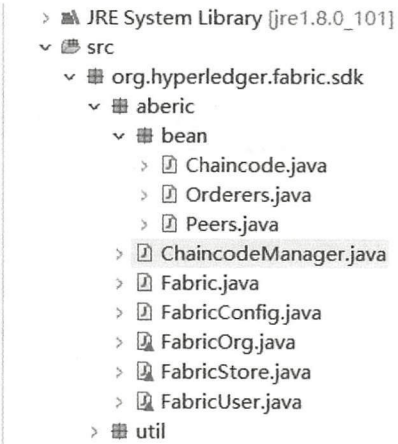


图9-1　SDK目录结构

Java-SDK 项目所需要导入的 jar 包如图 9-2 所示。

| 名称 | 修改日期 | 类型 | 大小 |
|---|---|---|---|
| bcpkix-jdk15on-1.58.jar | 2017/9/4 16:25 | JAR 文件 | 768 KB |
| bcprov-ext-jdk15on-1.58.jar | 2017/9/11 21:16 | JAR 文件 | 3,932 KB |
| commons-compress-1.14.jar | 2017/9/6 11:24 | JAR 文件 | 518 KB |
| commons-io-1.3.2.jar | 2017/10/8 17:08 | JAR 文件 | 86 KB |
| commons-io-2.6.jar | 2017/10/15 12:00 | JAR 文件 | 210 KB |
| commons-logging-1.2.jar | 2017/4/24 10:20 | JAR 文件 | 61 KB |
| fabric-sdk-java-1.0.1.jar | 2017/10/11 14:46 | JAR 文件 | 1,417 KB |
| fabric-sdk-java-1.0.1-sources.jar | 2017/10/11 14:51 | JAR 文件 | 424 KB |
| grpc-context-1.6.1.jar | 2017/9/13 10:12 | JAR 文件 | 21 KB |
| grpc-core-1.6.1.jar | 2017/9/8 11:29 | JAR 文件 | 472 KB |
| grpc-netty-1.6.1.jar | 2017/9/6 11:09 | JAR 文件 | 174 KB |
| grpc-protobuf-1.6.1.jar | 2017/9/6 11:09 | JAR 文件 | 6 KB |
| grpc-protobuf-lite-1.6.1.jar | 2017/9/13 10:07 | JAR 文件 | 8 KB |
| grpc-stub-1.6.1.jar | 2017/9/6 11:08 | JAR 文件 | 37 KB |
| guava-23.0.jar | 2017/9/8 11:24 | JAR 文件 | 2,554 KB |
| httpclient-4.5.3.jar | 2017/9/6 11:26 | JAR 文件 | 731 KB |
| httpcore-4.4.6.jar | 2017/1/7 14:49 | JAR 文件 | 317 KB |
| instrumentation-api-0.4.3.jar | 2017/9/13 9:56 | JAR 文件 | 92 KB |
| netty-all-4.1.9.Final.jar | 2017/10/12 11:24 | JAR 文件 | 3,429 KB |
| netty-all-4.1.9.Final-sources.jar | 2017/10/12 11:24 | JAR 文件 | 2,763 KB |
| org.apache.commons.codec-1.9.0.v2... | 2017/9/8 13:31 | JAR 文件 | 285 KB |
| protobuf-java-3.4.0.jar | 2017/9/4 16:35 | JAR 文件 | 1,352 KB |

图9-2　jar包

如果是用的 maven，pom 的内容参考如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>demo</groupId>
    <artifactId>aberic</artifactId>
    <version>1.0-SNAPSHOT</version>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.7.0</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <useUniqueVersions>false</useUniqueVersions>
                            <classpathPrefix>lib/</classpathPrefix>
                            <mainClass>org.hyperledger.fabric.sdk.Main</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <properties>
        <bouncycastle>1.59</bouncycastle>
        <grpc>1.6.1</grpc>
        <slf4j>1.7.25</slf4j>
```

```
        <log4j>2.10.0</log4j>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>${log4j}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>${log4j}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-slf4j-impl</artifactId>
            <version>${log4j}</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>${slf4j}</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>${slf4j}</version>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.bouncycastle</groupId>
            <artifactId>bcpkix-jdk15on</artifactId>
            <version>${bouncycastle}</version>
        </dependency>
        <dependency>
            <groupId>org.bouncycastle</groupId>
            <artifactId>bcprov-ext-jdk15on</artifactId>
            <version>${bouncycastle}</version>
        </dependency>

        <dependency>
            <groupId>io.grpc</groupId>
```



```
        <artifactId>grpc-context</artifactId>
        <version>${grpc}</version>
    </dependency>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-core</artifactId>
        <version>${grpc}</version>
    </dependency>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-netty</artifactId>
        <version>${grpc}</version>
    </dependency>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-protobuf</artifactId>
        <version>${grpc}</version>
    </dependency>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-protobuf-lite</artifactId>
        <version>${grpc}</version>
    </dependency>
    <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-stub</artifactId>
        <version>${grpc}</version>
    </dependency>

    <dependency>
        <groupId>com.google.guava</groupId>
        <artifactId>guava</artifactId>
        <version>23.0</version>
    </dependency>

    <dependency>
        <groupId>org.apache.httpcomponents</groupId>
        <artifactId>httpclient</artifactId>
        <version>4.5.5</version>
    </dependency>

    <dependency>
        <groupId>org.apache.httpcomponents</groupId>
        <artifactId>httpcore</artifactId>
```

```
            <version>4.4.9</version>
        </dependency>

        <dependency>
            <groupId>com.google.instrumentation</groupId>
            <artifactId>instrumentation-api</artifactId>
            <version>0.4.3</version>
        </dependency>

        <dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
            <version>4.1.9.Final</version>
        </dependency>

        <dependency>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
            <version>1.11</version>
        </dependency>

        <dependency>
            <groupId>com.google.protobuf</groupId>
            <artifactId>protobuf-java</artifactId>
            <version>3.5.1</version>
        </dependency>

        <dependency>
            <groupId>org.hyperledger.fabric-sdk-java</groupId>
            <artifactId>fabric-sdk-java</artifactId>
            <version>1.0.1</version>
        </dependency>
    </dependencies>

</project>
```

当上述 jar 或 pom 已经准备完成时，接下来就可以进入 SDK 开发了。

## 9.2 SDK代码使用

在开始编写使用 SDK 之前，首先需要对 Fabric 的事务流程有一个回顾，在第 6.2 节中介绍过客户端如何处理 Fabric 网络中的请求，这里结合 Java-SDK 再次阐述一遍。



Java-SDK 也是客户端的一种体现，当 SDK 发起一个事务 invoke 请求时，会将事务请求参数发送至 Peer 节点服务器，随后 Peer 节点服务器对其参数内容进行校验以及背书校验。如果校验成功，则返回处理成功的消息给客户端，在该消息中包含了本次请求的读写集。这一段包含读写集的消息将会被客户端发送给 Orderer 排序服务器，此时 Orderer 排序服务器再收到消息后进行创建区块等操作。待最终排序完成则执行广播操作，将所生成的区块广播到各个 Peer 节点服务器。

根据上述事务描述，再来看 SDK 如何使用。

向 Peer 发送请求则需要一个 Peers 对象，因为 Peer 可能是多个；随后将读写集发送给 Orderer 排序服务节点，则需要一个 Orderers 对象，因为 Orderer 也可能是多个。而它们之间的请求都是通过智能合约来实现的，所以需要一个 Chaincode 对象。

有了上述最基本的三个对象，就可以根据 SDK 中提供的 Demo 来慢慢梳理其后面的关键内容了。

### 9.2.1　Orderers对象

在 Orderers 对象中定义了两个成员变量，如下所示：

- ordererDomainName（String），指出了排序服务器所在的根域名。
- Orderers（List<Orderer>），是一个排序服务器队列。

内部还有一个名为 Orderer 的内部类，Orderer 类也有两个成员变量，如下所示：

- ordererName（String），排序服务器的域名。
- ordererLocation（String），排序服务器的访问地址，包括域名或 IP:PORT 组合。

具体类源码如下所示：

```
public class Orderers {

    /** orderer 排序服务器所在根域名 */
    private String ordererDomainName;
    /** orderer 排序服务器集合 */
    private List<Orderer> orderers;

    public Orderers() {
        orderers = new ArrayList<>();
    }

    public String getOrdererDomainName() {
        return ordererDomainName;
    }
```

```
    public void setOrdererDomainName(String ordererDomainName) {
        this.ordererDomainName = ordererDomainName;
    }

    /** 新增排序服务器 */
    public void addOrderer(String name, String location) {
        orderers.add(new Orderer(name, location));
    }

    /** 获取排序服务器集合 */
    public List<Orderer> get() {
        return orderers;
    }

    /**
     * 排序服务器对象
     *
     * @author 杨毅
     *
     * @date 2017年10月18日 - 下午2:06:22
     * @email abericyang@gmail.com
     */
    public class Orderer {

        /** orderer 排序服务器的域名 */
        private String ordererName;
        /** orderer 排序服务器的访问地址 */
        private String ordererLocation;

        public Orderer(String ordererName, String ordererLocation) {
            super();
            this.ordererName = ordererName;
            this.ordererLocation = ordererLocation;
        }

        public String getOrdererName() {
            return ordererName;
        }

        public void setOrdererName(String ordererName) {
            this.ordererName = ordererName;
        }

        public String getOrdererLocation() {
```



```
            return ordererLocation;
        }

        public void setOrdererLocation(String ordererLocation) {
            this.ordererLocation = ordererLocation;
        }

    }

}
```

其中，排序服务器的根域名可以帮助 SDK 定位排序服务器组所属配置文件的路径，而排序服务器域名则用于定位单台排序服务器的配置文件路径。

## 9.2.2　Peers对象

（1）在 Peers 对象中定义了 4 个成员变量：

- orgName（String），当前指定的组织名称。
- orgMSPID（String），当前指定的组织 ID 名。
- orgDomainName（String），当前指定的组织所在根域名。
- peers（List< Peer>），是一个 Peer 节点服务器队列。

（2）内部还有一个名为 Peer 的内部类，Peer 类有 6 个成员变量：

- peerName（String），当前指定的组织节点名称。
- peerEventHubName（String），当前指定的组织节点事件域名。
- peerLocation（String），当前指定的组织节点访问地址。
- peerEventHubLocation（String），当前指定的组织节点事件监听访问地址。
- caLocation（String），当前指定的组织节点 ca 访问地址。
- addEventHub（boolean），当前 peer 是否增加 Event 事件处理。

具体类源码如下所示：

```
public class Peers {

    /** 当前指定的组织名称 */
    private String orgName; // Org1
    /** 当前指定的组织名称 */
    private String orgMSPID; // Org1MSP
    /** 当前指定的组织所在根域名 */
    private String orgDomainName; // org1.example.com
    /** orderer 排序服务器集合 */
    private List<Peer> peers;
```

```
    public Peers() {
        peers = new ArrayList<>();
    }

    public String getOrgName() {
        return orgName;
    }

    public void setOrgName(String orgName) {
        this.orgName = orgName;
    }

    public String getOrgMSPID() {
        return orgMSPID;
    }

    public void setOrgMSPID(String orgMSPID) {
        this.orgMSPID = orgMSPID;
    }

    public String getOrgDomainName() {
        return orgDomainName;
    }

    public void setOrgDomainName(String orgDomainName) {
        this.orgDomainName = orgDomainName;
    }

    /** 新增排序服务器 */
    public void addPeer(String peerName, String peerEventHubName, String
peerLocation, String peerEventHubLocation, String caLocation) {
        peers.add(new Peer(peerName, peerEventHubName, peerLocation,
peerEventHubLocation, caLocation));
    }

    /** 获取排序服务器集合 */
    public List<Peer> get() {
        return peers;
    }

    /**
     * 节点服务器对象
     *
```



```
 * @author 杨毅
 *
 * @date 2017年11月11日 - 下午6:56:14
 * @email abericyang@gmail.com
 */
public class Peer {

    /** 当前指定的组织节点域名 */
    private String peerName; // peer0.org1.example.com
    /** 当前指定的组织节点事件域名 */
    private String peerEventHubName; // peer0.org1.example.com
    /** 当前指定的组织节点访问地址 */
    private String peerLocation; // grpc://110.131.116.21:7051
    /** 当前指定的组织节点事件监听访问地址 */
    private String peerEventHubLocation; // grpc://110.131.116.21:7053
    /** 当前指定的组织节点ca访问地址 */
    private String caLocation; // http://110.131.116.21:7054
    /** 当前peer是否增加Event事件处理 */
    private boolean addEventHub = false;

    public Peer(String peerName, String peerEventHubName, String peerLocation,
String peerEventHubLocation, String caLocation) {
        this.peerName = peerName;
        this.peerEventHubName = peerEventHubName;
        this.peerLocation = peerLocation;
        this.peerEventHubLocation = peerEventHubLocation;
        this.caLocation = caLocation;
    }

    public String getPeerName() {
        return peerName;
    }

    public void setPeerName(String peerName) {
        this.peerName = peerName;
    }

    public String getPeerEventHubName() {
        return peerEventHubName;
    }

    public void setPeerEventHubName(String peerEventHubName) {
        this.peerEventHubName = peerEventHubName;
    }
```

```
        public String getPeerLocation() {
            return peerLocation;
        }

        public void setPeerLocation(String peerLocation) {
            this.peerLocation = peerLocation;
        }

        public String getPeerEventHubLocation() {
            return peerEventHubLocation;
        }

        public void setPeerEventHubLocation(String eventHubLocation) {
            this.peerEventHubLocation = eventHubLocation;
        }

        public String getCaLocation() {
            return caLocation;
        }

        public void setCaLocation(String caLocation) {
            this.caLocation = caLocation;
        }

        public boolean isAddEventHub() {
            return addEventHub;
        }

        public void addEventHub(boolean addEventHub) {
            this.addEventHub = addEventHub;
        }

    }

}
```

其中，绝大多数成员变量与 Orderer 排序服务对象类似，都与其在服务器 YAML 配置文件中的环境变量相对应。而在 SDK 的主要用途也与 YAML 配置文件中路径映射的目的一样，便于定位所需验证文件的路径。

### 9.2.3 Chaincode对象

在 Chaincode 对象定义了 6 个成员变量，如下所示：



- channelName（String），当前将要访问的智能合约所属频道名称。
- chaincodeName（String），智能合约名称。
- chaincodePath（String），智能合约在服务器中的安装路径。
- chaincodeVersion（String），智能合约版本号。
- transactionWaitTime（int），临时变量控制等待部署和调用的时间，以完成在发出之前的事件。当 SDK 能够接收来自于此的事件时，这个问题就会被删除。
- deployWatiTime（int），临时变量控制等待部署和调用的时间，以完成在发出之前的事件。当 SDK 能够接收来自于此的事件时，这个问题就会被删除。

具体类源码如下所示：

```
public class Chaincode {

    /** 当前将要访问的智能合约所属频道名称 */
    private String channelName; // ffetest
    /** 智能合约名称 */
    private String chaincodeName; // ffetestcc
    /** 智能合约安装路径 */
    private String chaincodePath; //
github.com/hyperledger/fabric/xxx/chaincode/go/example/test
    /** 智能合约版本号 */
    private String chaincodeVersion; // 1.0
    /** 临时变量控制等待部署和调用的时间，以完成在发出之前的事件。当SDK能够接收来自于此的事
    件时，这个问题就会被删除 */
    private int transactionWaitTime = 100000;
    /** 临时变量控制等待部署和调用的时间，以完成在发出之前的事件。当SDK能够接收来自于此的事
    件时，这个问题就会被删除 */
    private int deployWatiTime = 120000;

    public String getChannelName() {
        return channelName;
    }

    public void setChannelName(String channelName) {
        this.channelName = channelName;
    }

    public String getChaincodeName() {
        return chaincodeName;
    }

    public void setChaincodeName(String chaincodeName) {
        this.chaincodeName = chaincodeName;
```

```
    }

    public String getChaincodePath() {
        return chaincodePath;
    }

    public void setChaincodePath(String chaincodePath) {
        this.chaincodePath = chaincodePath;
    }

    public String getChaincodeVersion() {
        return chaincodeVersion;
    }

    public void setChaincodeVersion(String chaincodeVersion) {
        this.chaincodeVersion = chaincodeVersion;
    }

    public int getTransactionWaitTime() {
        return transactionWaitTime;
    }

    public void setTransactionWaitTime(int invokeWatiTime) {
        this.transactionWaitTime = invokeWatiTime;
    }

    public int getDeployWaitTime() {
        return deployWatiTime;
    }

    public void setDeployWaitTime(int deployWaitTime) {
        this.deployWatiTime = deployWaitTime;
    }

}
```

关于智能合约安装路径的变量，这里指的是智能合约在安装时所输入的安装路径。

### 9.2.4 FabricUser

每一个 Peer 节点服务器都对应了一个组织成员下的会员对象，即拥有合法身份访问联盟数据的对象。这个对象中的密钥文件由 CA 生成，以第 6 章为例，其所对应的密钥文件目录分别是 /home/docker/github.com/hyperledger/fabric/aberic/crypto-config/peerOrganizations/org2. 这些

example.com/users/Admin@org2.example.com/msp/keystore 以及 /home/docker/github.com/hyperledger/fabric/aberic/crypto-config/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp/signcerts，即 keyfile 和 certfile。

这个会员也将从属于一个 Peer 节点，及 Peer 节点所从属的组织，即 FabricUser 对象也会包含组织及单节点相关信息，具体源码如下所示：

```
class FabricUser implements User, Serializable {

    private static final long serialVersionUID = 5695080465408336815L;

    /** 名称 */
    private String name;
    /** 账户 */
    private String account;
    /** 从属联盟 */
    private String affiliation;
    /** 组织 */
    private String organization;
    /** 会员ID */
    private String mspId;
    /** 注册登记操作 */
    Enrollment enrollment = null;

    /** 存储配置对象 */
    private transient FabricStore keyValStore;
    private String keyValStoreName;

    public FabricUser(String name, String org, FabricStore store) {
        this.name = name;
        this.keyValStore = store;
        this.organization = org;
        this.keyValStoreName = toKeyValStoreName(this.name, org);

        String memberStr = keyValStore.getValue(keyValStoreName);
        if (null != memberStr) {
            saveState();
        } else {
            restoreState();
        }
    }

    /**
     * 设置账户信息并将用户状态更新至存储配置对象
```

```
     *
     * @param account
     *          账户
     */
    public void setAccount(String account) {
        this.account = account;
        saveState();
    }

    @Override
    public String getAccount() {
        return this.account;
    }

    /**
     * 设置从属联盟信息并将用户状态更新至存储配置对象
     *
     * @param affiliation
     *          从属联盟
     */
    public void setAffiliation(String affiliation) {
        this.affiliation = affiliation;
        saveState();
    }

    @Override
    public String getAffiliation() {
        return this.affiliation;
    }

    @Override
    public Enrollment getEnrollment() {
        return this.enrollment;
    }

    /**
     * 设置会员ID信息并将用户状态更新至存储配置对象
     *
     * @param mspID
     *          会员ID
     */
    public void setMspId(String mspID) {
        this.mspId = mspID;
        saveState();
```

```
    }

    @Override
    public String getMspId() {
        return this.mspId;
    }

    @Override
    public String getName() {
        return this.name;
    }

    /**
     * 设置注册登记操作信息并将用户状态更新至存储配置对象
     *
     * @param enrollment
     *            注册登记操作
     */
    public void setEnrollment(Enrollment enrollment) {
        this.enrollment = enrollment;
        saveState();
    }

    /**
     * 确定这个名称是否已注册
     *
     * @return 与否
     */
    public boolean isRegistered() {
        return !StringUtil.isNullOrEmpty(enrollmentSecret);
    }

    /**
     * 确定这个名字是否已经注册
     *
     * @return 与否
     */
    public boolean isEnrolled() {
        return this.enrollment != null;
    }

    /** 将用户状态保存至存储配置对象 */
    public void saveState() {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
```

```
            try {
                ObjectOutputStream oos = new ObjectOutputStream(bos);
                oos.writeObject(this);
                oos.flush();
                keyValStore.setValue(keyValStoreName, Hex.toHexString(bos.toByteArray
()));
                bos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        /**
         * 从键值存储中恢复该用户的状态(如果找到的话)。如果找不到，什么也不要做
         *
         * @return 返回用户
         */
        private FabricUser restoreState() {
            String memberStr = keyValStore.getValue(keyValStoreName);
            if (null != memberStr) {
                // 用户在键值存储中被找到，因此恢复状态
                byte[] serialized = Hex.decode(memberStr);
                ByteArrayInputStream bis = new ByteArrayInputStream(serialized);
                try {
                    ObjectInputStream ois = new ObjectInputStream(bis);
                    FabricUser state = (FabricUser) ois.readObject();
                    if (state != null) {
                        this.name = state.name;
                        this.roles = state.roles;
                        this.account = state.account;
                        this.affiliation = state.affiliation;
                        this.organization = state.organization;
                        this.enrollment = state.enrollment;
                        this.mspId = state.mspId;
                        return this;
                    }
                } catch (Exception e) {
                    throw new RuntimeException(String.format("Could not restore state of
member %s", this.name), e);
                }
            }
            return null;
        }
```

```
    public static String toKeyValStoreName(String name, String org) {
        System.out.println("toKeyValStoreName = " + "user." + name + org);
        return "user." + name + org;
    }

}
```

如源码所示，其中包含了一个名为 FabricStore 的成员变量，FabricStore 变量将获取缓存配置文件中的会员信息，而信息是否为空将决定初始化该会员对象时是否执行更新操作。

### 9.2.5　FabricStore

FabricStore 对象的主要作用是根据用户初始化时的资源来做一次缓存操作，缓存于当前系统的缓存目录下。如果是 Linux 系统，则缓存于/tmp 目录下，如果是 Windows 系统，则缓存于 C:\Users\登录用户名\AppData\Local\Temp 目录下。

在执行缓存的同时，会在该缓存目录下创建一个名为 HFCSampletest.properties 的缓存文件，每当执行 Fabric 网络事务时，都会尝试读取该缓存并继续后续操作。该对象与 FabricUser 配合使用，其具体源码如下：

```
class FabricStore {

    private String file;
    /** 用户信息集合 */
    private final Map<String, FabricUser> members = new HashMap<>();

    public FabricStore(File file) {
        this.file = file.getAbsolutePath();
    }

    /**
     * 设置与名称相关的值
     *
     * @param name
     *            名称
     * @param value
     *            相关值
     */
    public void setValue(String name, String value) {
        Properties properties = loadProperties();
        try (OutputStream output = new FileOutputStream(file)) {
            properties.setProperty(name, value);
            properties.store(output, "");
            output.close();
```

```
            } catch (IOException e) {
                System.out.println(String.format("Could not save the keyvalue store,
reason:%s", e.getMessage()));
            }
        }

        /**
         * 获取与名称相关的值
         *
         * @param 名称
         * @return 相关值
         */
        public String getValue(String name) {
            Properties properties = loadProperties();
            return properties.getProperty(name);
        }

        /**
         * 加载配置文件
         *
         * @return 配置文件对象
         */
        private Properties loadProperties() {
            Properties properties = new Properties();
            try (InputStream input = new FileInputStream(file)) {
                properties.load(input);
                input.close();
            } catch (FileNotFoundException e) {
                System.out.println(String.format("Could not find the file \"%s\"",
file));
            } catch (IOException e) {
                System.out.println(String.format("Could not load keyvalue store from
file \"%s\", reason:%s", file, e.getMessage()));
            }
            return properties;
        }

        /**
         * 用给定的名称获取用户
         *
         * @param 名称
         * @param 组织
         *
         * @return 用户
```

```
        */
       public FabricUser getMember(String name, String org) {
           // 尝试从缓存中获取User状态
           FabricUser fabricUser = members.get(FabricUser.toKeyValStoreName(name, org));
           if (null != fabricUser) {
               return fabricUser;
           }
           // 创建User，并尝试从键值存储中恢复它的状态(如果找到的话)
           fabricUser = new FabricUser(name, org, this);
           return fabricUser;
       }

       /**
        * 用给定的名称获取用户
        *
        * @param name
        *          名称
        * @param org
        *          组织
        * @param mspId
        *          会员ID
        * @param privateKeyFile
        * @param certificateFile
        *
        * @return user 用户
        *
        * @throws IOException
        * @throws NoSuchAlgorithmException
        * @throws NoSuchProviderException
        * @throws InvalidKeySpecException
        */
       public FabricUser getMember(String name, String org, String mspId, File
privateKeyFile, File certificateFile)
               throws IOException, NoSuchAlgorithmException, NoSuchProviderException,
InvalidKeySpecException {
           try {
               // 尝试从缓存中获取User状态
               FabricUser fabricUser = members.get(FabricUser.toKeyValStoreName(name,
org));
               if (null != fabricUser) {
                   System.out.println("尝试从缓存中获取User状态 User = " + fabricUser);
                   return fabricUser;
               }
               // 创建User，并尝试从键值存储中恢复它的状态(如果找到的话)
```

```
            fabricUser = new FabricUser(name, org, this);
            fabricUser.setMspId(mspId);
            String certificate = new String(IOUtils.toByteArray(new FileInputStream
(certificateFile)), "UTF-8");
            PrivateKey privateKey = getPrivateKeyFromBytes(IOUtils.toByteArray(new
FileInputStream(privateKeyFile)));
            fabricUser.setEnrollment(new StoreEnrollement(privateKey, certificate));
            return fabricUser;
         } catch (IOException e) {
            e.printStackTrace();
            throw e;
         } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            throw e;
         } catch (NoSuchProviderException e) {
            e.printStackTrace();
            throw e;
         } catch (InvalidKeySpecException e) {
            e.printStackTrace();
            throw e;
         } catch (ClassCastException e) {
            e.printStackTrace();
            throw e;
         }
      }

      /**
       * 通过字节数组信息获取私钥
       *
       * @param data
       *           字节数组
       *
       * @return 私钥
       *
       * @throws IOException
       * @throws NoSuchProviderException
       * @throws NoSuchAlgorithmException
       * @throws InvalidKeySpecException
       */
      private PrivateKey getPrivateKeyFromBytes(byte[] data) throws IOException,
NoSuchProviderException, NoSuchAlgorithmException, InvalidKeySpecException {
         final Reader pemReader = new StringReader(new String(data));
         final PrivateKeyInfo pemPair;
         try (PEMParser pemParser = new PEMParser(pemReader)) {
```

```
            pemPair = (PrivateKeyInfo) pemParser.readObject();
        }
        PrivateKey privateKey = new
JcaPEMKeyConverter().setProvider(BouncyCastleProvider.PROVIDER_NAME).getPrivateKey(pem
Pair);
        return privateKey;
    }

    static {
        try {
            Security.addProvider(new BouncyCastleProvider());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 自定义注册登记操作类
     *
     * @author yangyi47
     *
     */
    static final class StoreEnrollement implements Enrollment, Serializable {

        private static final long serialVersionUID = 6965341351799577442L;

        /** 私钥 */
        private final PrivateKey privateKey;
        /** 授权证书 */
        private final String certificate;

        StoreEnrollement(PrivateKey privateKey, String certificate) {
            this.certificate = certificate;
            this.privateKey = privateKey;
        }

        @Override
        public PrivateKey getKey() {
            return privateKey;
        }

        @Override
        public String getCert() {
            return certificate;
```

```
        }
    }

}
```

该对象对于性能有所帮助，所缓存的内容也会在服务暂停后由所在系统不定时销毁，对安全性也有一定的帮助。

### 9.2.6 FabricOrg

FabricOrg 是一个联盟的单个组织对象，按照官方的说法可以是医院、机构、学校、事业单位、公司和社团等。

一个 Fabric 网络是由一个或多个组织共同组成的，每一个组织都具备自身独立性，既可以选择加入，也可以选择退出，是 Fabric 大型网络中最大的单元，也具有最高的独立性。

一个组织可以由 0 个以上的排序服务节点和 1 个以上的 Peer 节点共同组成，如果当前组织中没有部署排序服务节点，则必须由联盟发起方提供一个共享排序服务节点，否则是无法加入该网络的。

所以，在 FabricOrg 对象中，将包含之前所编写的 Orderers 和 Peers 对象，而 Chaincode 对象相对独立，只运行于 Channel 上。

当 FabricOrg 对象被构造的时候，会分别将 Orderers 和 Peers 中的队列内容一并读取并初始化，而 FabricOrg 对象也将作为 SDK 输出组织信息的统一出口。

其具体源码如下所示：

```
class FabricOrg {

    private static Logger log = LoggerFactory.getLogger(FabricOrg.class);

    /** 名称 */
    private String name;
    /** 会员ID */
    private String mspid;
    /** ca 客户端 */
    private HFCAClient caClient;

    /** 用户集合 */
    Map<String, User> userMap = new HashMap<>();
    /** 本地节点集合 */
    Map<String, String> peerLocations = new HashMap<>();
    /** 本地排序服务集合 */
    Map<String, String> ordererLocations = new HashMap<>();
```

```
        /** 本地事件集合 */
        Map<String, String> eventHubLocations = new HashMap<>();
        /** 节点集合 */
        Set<Peer> peers = new HashSet<>();
        /** 联盟管理员用户 */
        private FabricUser admin;
        /** 本地 ca */
        private String caLocation;
        /** ca 配置 */
        private Properties caProperties = null;

        /** 联盟单节点管理员用户 */
        private FabricUser peerAdmin;

        /** 域名名称 */
        private String domainName;

        public FabricOrg(String username, Peers peers, Orderers orderers, FabricStore
fabricStore, String cryptoConfigPath)
                throws NoSuchAlgorithmException, NoSuchProviderException,
InvalidKeySpecException, IOException {
            this.name = peers.getOrgName();
            this.mspid = peers.getOrgMSPID();
            for (int i = 0; i < peers.get().size(); i++) {
                addPeerLocation(peers.get().get(i).getPeerName(), peers.get().get(i).
getPeerLocation());
                addEventHubLocation(peers.get().get(i).getPeerEventHubName(),
peers.get().get(i).getPeerEventHubLocation());
                setCALocation(peers.get().get(i).getCaLocation());
            }
            for (int i = 0; i < orderers.get().size(); i++) {
                addOrdererLocation(orderers.get().get(i).getOrdererName(),
orderers.get().get(i).getOrdererLocation());
            }
            setDomainName(peers.getOrgDomainName()); // domainName=tk.anti-moth.com

            // Set up HFCA for Org1
            // setCAClient(HFCAClient.createNewInstance(peers.getCaLocation(),
getCAProperties()));

            // setAdmin(fabricStore.getMember("admin", peers.getOrgName())); // 设置该组
织的管理员

            File skFile = Paths.get(cryptoConfigPath, "/peerOrganizations/",
```

```
peers.getOrgDomainName(), String.format("/users/%s@%s/msp/keystore", username,
peers.getOrgDomainName())).toFile();
          File certificateFile = Paths.get(cryptoConfigPath, "/peerOrganizations/",
peers.getOrgDomainName(),
                  String.format("/users/%s@%s/msp/signcerts/%s@%s-cert.pem", username,
peers.getOrgDomainName(), username, peers.getOrgDomainName())).toFile();
          log.debug("skFile = " + skFile.getAbsolutePath());
          log.debug("certificateFile = " + certificateFile.getAbsolutePath());
          setPeerAdmin(fabricStore.getMember(peers.getOrgName() + username,
peers.getOrgName(), peers.getOrgMSPID(), findFileSk(skFile), certificateFile)); // 一个
特殊的用户，可以创建通道，连接对等点，并安装链码
       }

       /**
        * 添加本地节点
        *
        * @param name
        *          节点key
        * @param location
        *          节点
        */
       public void addPeerLocation(String name, String location) {
          peerLocations.put(name, location);
       }

       /**
        * 添加本地组织
        *
        * @param name
        *          组织key
        * @param location
        *          组织
        */
       public void addOrdererLocation(String name, String location) {
          ordererLocations.put(name, location);
       }

       /**
        * 添加本地事件
        *
        * @param name
        *          事件key
        * @param location
        *          事件
```

```
     */
    public void addEventHubLocation(String name, String location) {
        eventHubLocations.put(name, location);
    }

    /**
     * 获取本地节点
     *
     * @param name
     *          节点key
     * @return 节点
     */
    public String getPeerLocation(String name) {
        return peerLocations.get(name);
    }

    /**
     * 获取本地组织
     *
     * @param name
     *          组织key
     * @return 组织
     */
    public String getOrdererLocation(String name) {
        return ordererLocations.get(name);
    }

    /**
     * 获取本地事件
     *
     * @param name
     *          事件key
     * @return 事件
     */
    public String getEventHubLocation(String name) {
        return eventHubLocations.get(name);
    }

    /**
     * 获取一个不可修改的本地节点key集合
     *
     * @return 节点key集合
     */
    public Set<String> getPeerNames() {
```

```
        return Collections.unmodifiableSet(peerLocations.keySet());
    }

    /**
     * 获取一个不可修改的本地节点集合
     *
     * @return 节点集合
     */
    public Set<Peer> getPeers() {
        return Collections.unmodifiableSet(peers);
    }

    /**
     * 获取一个不可修改的本地组织key集合
     *
     * @return 组织key集合
     */
    public Set<String> getOrdererNames() {
        return Collections.unmodifiableSet(ordererLocations.keySet());
    }

    /**
     * 获取一个不可修改的本地组织集合
     *
     * @return 组织集合
     */
    public Collection<String> getOrdererLocations() {
        return Collections.unmodifiableCollection(ordererLocations.values());
    }

    /**
     * 获取一个不可修改的本地事件key集合
     *
     * @return 事件key集合
     */
    public Set<String> getEventHubNames() {
        return Collections.unmodifiableSet(eventHubLocations.keySet());
    }

    /**
     * 获取一个不可修改的本地事件集合
     *
     * @return 事件集合
     */
```

```
public Collection<String> getEventHubLocations() {
    return Collections.unmodifiableCollection(eventHubLocations.values());
}

/**
 * 向用户集合中添加用户
 *
 * @param user
 *          用户
 */
public void addUser(FabricUser user) {
    userMap.put(user.getName(), user);
}

/**
 * 从用户集合根据名称获取用户
 *
 * @param name
 *          名称
 * @return 用户
 */
public User getUser(String name) {
    return userMap.get(name);
}

/**
 * 向节点集合中添加节点
 *
 * @param peer
 *          节点
 */
public void addPeer(Peer peer) {
    peers.add(peer);
}

set/get等方法

/**
 * 从指定路径中获取后缀为 _sk 的文件，且该路径下有且仅有该文件
 *
 * @param directorys
 *          指定路径
 * @return File
 */
```

```
        private File findFileSk(File directory) {
            File[] matches = directory.listFiles((dir, name) -> name.endsWith("_sk"));
            if (null == matches) {
                throw new RuntimeException(String.format("Matches returned null does %s
directory exist?", directory.getAbsoluteFile().getName()));
            }
            if (matches.length != 1) {
                throw new RuntimeException(String.format("Expected in %s only 1 sk file
but found %d", directory.getAbsoluteFile().getName(), matches.length));
            }
            return matches[0];
        }

    }
```

FabricOrg 对象最终将被 ChaincodeManager 对象调用，是 SDK 中需要重点关注的一个对象。

### 9.2.7 FabricConfig

FabricConfig 是一个较为简单的配置对象，其主要用来辅助 ChaincodeManager 对象管理其中请求所需的配置和参数，具体源码如下：

```
    public class FabricConfig {

        private static Logger log = LoggerFactory.getLogger(FabricConfig.class);

        /** 节点服务器对象 */
        private Peers peers;
        /** 排序服务器对象 */
        private Orderers orderers;
        /** 智能合约对象 */
        private Chaincode chaincode;
        /** channel-artifacts所在路径：默认channel-artifacts所在路径/xxx/WEB-
INF/classes/fabric/channel-artifacts/ */
        private String channelArtifactsPath;
        /** crypto-config所在路径：默认crypto-config所在路径/xxx/WEB-INF/classes/fabric/
crypto-config/ */
        private String cryptoConfigPath;
        private boolean registerEvent = false;

        public FabricConfig() {
            // 默认channel-artifacts所在路径 /xxx/WEB-INF/classes/fabric/channel-
artifacts/
```

```
            channelArtifactsPath = getChannlePath() + "/channel-artifacts/";
            // 默认crypto-config所在路径 /xxx/WEB-INF/classes/fabric/crypto-config/
            cryptoConfigPath = getChannlePath() + "/crypto-config/";
        }

        /**
         * 默认Fabric配置路径
         *
         * @return D:/installSoft/apache-tomcat-9.0.0.M21-02/webapps/xxx/WEB-
INF/classes/fabric/channel-artifacts/
         */
        private String getChannlePath() {
            String directorys = FabricConfig.class.getClassLoader().getResource
("fabric").getFile();
            log.debug("directorys = " + directorys);
            File directory = new File(directorys);
            log.debug("directory = " + directory.getPath());

            return directory.getPath();
            // return "src/main/resources/fabric/channel-artifacts/";
        }

        set/get等方法

    }
```

在 SDK 实际的使用过程中，FabricConfig 也将会被提到前台调用，即应用层直接创建一个 FabricConfig 对象提供给 Manager 对象使用。

## 9.2.8　ChaincodeManager

ChaincodeManager 对象是整个 SDK 整理过程中的核心类，它主要承载了 Channel 以及 Chaincode 的初始化。随后，可以通过初始化的 Chaincode 执行智能合约的 invoke 及 query 方法，实现 SDK 最重要的两个 API。

在 ChaincodeManager 对象中，使用的对象均来自前面几节所述对象，所以注释没有写很全。如果读者朋友需要的话，可以自行补齐注释，且第 9.2 节中的所有对象都会打包提供一个完成的工程下载。本书只对其中的作用做出说明，且最终实现业务层内容的时候已经完全脱离 SDK 中的操作了。

具体的 ChaincodeManager 对象源码如下：

```
    public class ChaincodeManager {
```

```
        private static Logger log = LoggerFactory.getLogger(ChaincodeManager.class);

        private FabricConfig config;
        private Orderers orderers;
        private Peers peers;
        private Chaincode chaincode;

        private HFClient client;
        private FabricOrg fabricOrg;
        private Channel channel;
        private ChaincodeID chaincodeID;

        public ChaincodeManager(String username, FabricConfig fabricConfig)
                throws CryptoException, InvalidArgumentException,
NoSuchAlgorithmException, NoSuchProviderException, InvalidKeySpecException,
IOException, TransactionException {
            this.config = fabricConfig;

            orderers = this.config.getOrderers();
            peers = this.config.getPeers();
            chaincode = this.config.getChaincode();

            client = HFClient.createNewInstance();
            log.debug("Create instance of HFClient");
            client.setCryptoSuite(CryptoSuite.Factory.getCryptoSuite());
            log.debug("Set Crypto Suite of HFClient");

            fabricOrg = getFabricOrg(username);
            channel = getChannel();
            chaincodeID = getChaincodeID();

            client.setUserContext(fabricOrg.getPeerAdmin()); // 也许是1.0.0测试版的Bug,
            只有节点管理员可以调用链码
        }

        private FabricOrg getFabricOrg(String username) throws NoSuchAlgorithmException,
NoSuchProviderException, InvalidKeySpecException, IOException {

            // java.io.tmpdir : C:\Users\yangyi47\AppData\Local\Temp\
            File storeFile = new File(System.getProperty("java.io.tmpdir") +
"/HFCSampletest.properties");
            FabricStore fabricStore = new FabricStore(storeFile);

            // Get Org1 from configuration
```

```
        FabricOrg fabricOrg = new FabricOrg(username, peers, orderers, fabricStore,
config.getCryptoConfigPath());
        log.debug("Get FabricOrg");
        return fabricOrg;
    }

    private Channel getChannel()
            throws NoSuchAlgorithmException, NoSuchProviderException,
InvalidKeySpecException, IOException, CryptoException, InvalidArgumentException,
TransactionException {
        client.setUserContext(fabricOrg.getPeerAdmin());
        return getChannel(fabricOrg, client);
    }

    private Channel getChannel(FabricOrg fabricOrg, HFClient client)
            throws NoSuchAlgorithmException, NoSuchProviderException,
InvalidKeySpecException, IOException, CryptoException, InvalidArgumentException,
TransactionException {
        Channel channel = client.newChannel(chaincode.getChannelName());
        log.debug("Get Chain " + chaincode.getChannelName());

        channel.setTransactionWaitTime(chaincode.getTransactionWaitTime());
        channel.setDeployWaitTime(chaincode.getDeployWaitTime());

        for (int i = 0; i < peers.get().size(); i++) {
            File peerCert = Paths.get(config.getCryptoConfigPath(),
"/peerOrganizations", peers.getOrgDomainName(), "peers",
peers.get().get(i).getPeerName(), "tls/server.crt")
                    .toFile();
            if (!peerCert.exists()) {
                throw new RuntimeException(
                        String.format("Missing cert file for: %s. Could not find at
location: %s", peers.get().get(i).getPeerName(), peerCert.getAbsolutePath()));
            }
            Properties peerProperties = new Properties();
            peerProperties.setProperty("pemFile", peerCert.getAbsolutePath());
            // ret.setProperty("trustServerCertificate", "true"); //testing
            // environment only NOT FOR PRODUCTION!
            peerProperties.setProperty("hostnameOverride", peers.getOrgDomainName());
            peerProperties.setProperty("sslProvider", "openSSL");
            peerProperties.setProperty("negotiationType", "TLS");
            // 在grpc的NettyChannelBuilder上设置特定选项
            peerProperties.put("grpc.ManagedChannelBuilderOption.
maxInboundMessageSize", 9000000);
```

```
            channel.addPeer(client.newPeer(peers.get().get(i).getPeerName(),
fabricOrg.getPeerLocation(peers.get().get(i).getPeerName()), peerProperties));
            if (peers.get().get(i).isAddEventHub()) {
                channel.addEventHub(
                        client.newEventHub(peers.get().get(i).getPeerEventHubName(),
fabricOrg.getEventHubLocation(peers.get().get(i).getPeerEventHubName()),
peerProperties));
            }
        }

        for (int i = 0; i < orderers.get().size(); i++) {
            File ordererCert = Paths.get(config.getCryptoConfigPath(),
"/ordererOrganizations", orderers.getOrdererDomainName(), "orderers",
orderers.get().get(i).getOrdererName(),
                    "tls/server.crt").toFile();
            if (!ordererCert.exists()) {
                throw new RuntimeException(
                        String.format("Missing cert file for: %s. Could not find at
location: %s", orderers.get().get(i).getOrdererName(), ordererCert.getAbsolutePath()));
            }
            Properties ordererProperties = new Properties();
            ordererProperties.setProperty("pemFile", ordererCert.getAbsolutePath());
            ordererProperties.setProperty("hostnameOverride",
orderers.getOrdererDomainName());
            ordererProperties.setProperty("sslProvider", "openSSL");
            ordererProperties.setProperty("negotiationType", "TLS");
            ordererProperties.put("grpc.ManagedChannelBuilderOption.
maxInboundMessageSize", 9000000);
            ordererProperties.setProperty("ordererWaitTimeMilliSecs", "300000");
            channel.addOrderer(
                    client.newOrderer(orderers.get().get(i).getOrdererName(),
fabricOrg.getOrdererLocation(orderers.get().get(i).getOrdererName()),
ordererProperties));
        }

        log.debug("channel.isInitialized() = " + channel.isInitialized());
        if (!channel.isInitialized()) {
            channel.initialize();
        }
        return channel;
    }

    private ChaincodeID getChaincodeID() {
        return
```

```
ChaincodeID.newBuilder().setName(chaincode.getChaincodeName()).setVersion(chaincode.ge
tChaincodeVersion()).setPath(chaincode.getChaincodePath()).build();
        }

        /**
         * 执行智能合约
         *
         * @param fcn
         *          方法名
         * @param args
         *          参数数组
         */
        public Map<String, String> invoke(String fcn, String[] args)
                throws InvalidArgumentException, ProposalException, InterruptedException,
ExecutionException, TimeoutException, NoSuchAlgorithmException,
NoSuchProviderException, InvalidKeySpecException, CryptoException,
TransactionException, IOException {
            Map<String, String> resultMap = new HashMap<>();

            Collection<ProposalResponse> successful = new LinkedList<>();
            Collection<ProposalResponse> failed = new LinkedList<>();

            /// Send transaction proposal to all peers
            TransactionProposalRequest transactionProposalRequest =
client.newTransactionProposalRequest();
            transactionProposalRequest.setChaincodeID(chaincodeID);
            transactionProposalRequest.setFcn(fcn);
            transactionProposalRequest.setArgs(args);

            Map<String, byte[]> tm2 = new HashMap<>();
            tm2.put("HyperLedgerFabric",
"TransactionProposalRequest:JavaSDK".getBytes(UTF_8));
            tm2.put("method", "TransactionProposalRequest".getBytes(UTF_8));
            tm2.put("result", ":)".getBytes(UTF_8));
            transactionProposalRequest.setTransientMap(tm2);

            long currentStart = System.currentTimeMillis();
            Collection<ProposalResponse> transactionPropResp =
channel.sendTransactionProposal(transactionProposalRequest, channel.getPeers());
            for (ProposalResponse response : transactionPropResp) {
                if (response.getStatus() == ProposalResponse.Status.SUCCESS) {
                    successful.add(response);
                } else {
                    failed.add(response);
```

```
            }
        }
        log.info("channel send transaction proposal time = " +
( System.currentTimeMillis() - currentStart));

        Collection<Set<ProposalResponse>> proposalConsistencySets =
SDKUtils.getProposalConsistencySets(transactionPropResp);
        if (proposalConsistencySets.size() != 1) {
            log.error("Expected only one set of consistent proposal responses but
got " + proposalConsistencySets.size());
        }

        if (failed.size() > 0) {
            ProposalResponse firstTransactionProposalResponse =
failed.iterator().next();
            log.error("Not enough endorsers for inspect:" + failed.size() + "
endorser error: " + firstTransactionProposalResponse.getMessage() + ". Was verified: "
                    + firstTransactionProposalResponse.isVerified());
            resultMap.put("code", "error");
            resultMap.put("data", firstTransactionProposalResponse.getMessage());
            return resultMap;
        } else {
            log.info("Successfully received transaction proposal responses.");
            ProposalResponse resp = transactionPropResp.iterator().next();
            log.debug("TransactionID: " + resp.getTransactionID());
            byte[] x = resp.getChaincodeActionResponsePayload();
            String resultAsString = null;
            if (x != null) {
                resultAsString = new String(x, "UTF-8");
            }
            log.info("resultAsString = " + resultAsString);
            channel.sendTransaction(successful);
            resultMap.put("code", "success");
            resultMap.put("data", resultAsString);
            resultMap.put("txid", resp.getTransactionID());
            return resultMap;
        }
    }

    /**
     * 查询智能合约
     *
     * @param fcn
     *          方法名
```

```
        * @param args
        *          参数数组
        */
       public Map<String, String> query(String fcn, String[] args) throws
InvalidArgumentException, ProposalException, NoSuchAlgorithmException,
NoSuchProviderException, InvalidKeySpecException, CryptoException,
TransactionException, IOException {
           Map<String, String> resultMap = new HashMap<>();
           String payload = "";
           QueryByChaincodeRequest queryByChaincodeRequest =
client.newQueryProposalRequest();
           queryByChaincodeRequest.setArgs(args);
           queryByChaincodeRequest.setFcn(fcn);
           queryByChaincodeRequest.setChaincodeID(chaincodeID);

           Map<String, byte[]> tm2 = new HashMap<>();
           tm2.put("HyperLedgerFabric",
"QueryByChaincodeRequest:JavaSDK".getBytes(UTF_8));
           tm2.put("method", "QueryByChaincodeRequest".getBytes(UTF_8));
           queryByChaincodeRequest.setTransientMap(tm2);

           Collection<ProposalResponse> queryProposals =
channel.queryByChaincode(queryByChaincodeRequest, channel.getPeers());
           for (ProposalResponse proposalResponse : queryProposals) {
               if (!proposalResponse.isVerified() || proposalResponse.getStatus() !=
ProposalResponse.Status.SUCCESS) {
                   log.debug("Failed query proposal from peer " +
proposalResponse.getPeer().getName() + " status: " + proposalResponse.getStatus() + ".
Messages: "
                           + proposalResponse.getMessage() + ". Was verified : " +
proposalResponse.isVerified());
                   resultMap.put("code", "error");
                   resultMap.put("data", "Failed query proposal from peer " +
proposalResponse.getPeer().getName() + " status: " + proposalResponse.getStatus() + ".
Messages: "
                           + proposalResponse.getMessage() + ". Was verified : " +
proposalResponse.isVerified());
               } else {
                   payload =
proposalResponse.getProposalResponse().getResponse().getPayload().toStringUtf8();
                   log.debug("Query payload from peer: " +
proposalResponse.getPeer().getName());
                   log.debug("TransactionID: " + proposalResponse.getTransactionID());
                   log.debug("" + payload);
```

```
                resultMap.put("code", "success");
                resultMap.put("data", payload);
                resultMap.put("txid", proposalResponse.getTransactionID());
            }
        }
        return resultMap;
    }
}
```

接下来，开始对本节所编写出的 SDK 中间件具体使用。

## 9.3 SDK使用方法

通过第 9.2 节的内容，已经可以组成一个 Fabric-Java-SDK 的出口，而调用该出口的对象需要在应用层进行指定。本书以第 7 章的环境作为 SDK 的测试环境，以 Spring MVC 作为 Web 接口提供项目方案。

首先需要一个管理器，用于应用层管理 SDK 的相关操作，本书中命名为 TestManager，具体源码如下：

```
public class TestManager extends BaseManager{

    private static TestManager instance;

    private ChaincodeManager abericChaincodeManager;

    public static TestManager obtain() throws CryptoException,
InvalidArgumentException, NoSuchAlgorithmException, NoSuchProviderException,
InvalidKeySpecException, TransactionException, IOException {
        if (null == instance) {
            synchronized (TestManager.class) {
                if (null == instance) {
                    instance = new TestManager();
                }
            }
        }
        return instance;
    }

    private TestManager() throws CryptoException, InvalidArgumentException,
NoSuchAlgorithmException, NoSuchProviderException, InvalidKeySpecException,
TransactionException,
    IOException {
```



```
            abericChaincodeManager = obtainDemoChaincodeManager();
        }

        public ChaincodeManager getDemoChaincodeManager() {
            return abericChaincodeManager;
        }

        private ChaincodeManager obtainDemoChaincodeManager() throws CryptoException,
InvalidArgumentException, NoSuchAlgorithmException, NoSuchProviderException,
InvalidKeySpecException, TransactionException, IOException {
            Fabric fabric = new Fabric();
            ChaincodeManager chaincodeManager = fabric
            .setUser("Admin")
            .setChannleArtifactsPath(getChannleArtifactsPath("aberic"))
            .setCryptoConfigPath(getCryptoConfigPath("aberic"))
            .setPeers("Org1", "Org1MSP", "org1.example.com")
            .addPeer("peer0.org1.example.com", "peer0.org1.example.com",
"grpc://47.104.202.224:7051", "grpc://47.104.202.224:7053", "http://47.104.202.
224:7054")
            .setOrderers("example.com")
            .addOrderer("orderer0.example.com", "grpc://47.104.73.77:7050")
            .setChaincode("mychannel", "mycc", "github.com/hyperledger/fabric/aberic/
chaincode/go/9fbank", "1.0")
            .getChaincodeManager();
            return chaincodeManager;
        }

    }
```

在 TestManager 中对 ChaincodeManager 进行了初始化，并最终得到了 ChaincodeManager 对象。在第 9.2.8 节中已经说过，ChaincodeManager 对象在初始化之后就可以直接操作于智能合约。

在获取 ChaincodeManager 对象之后，就需要在业务层开始执行合约的相关操作。通过第 7 章的内容，智能合约所接收的参数是一个参数数组，而数组的第一个参数被当作方法名使用。在 Java-SDK 中更加细化了这一步，ChaincodeManager 对象将执行的方法会传入一个方法名和一个纯粹的参数数组（不带方法名的数组）。

这里不再赘述 JavaWeb 相关知识，直接进入 TestServiceImpl 业务层。

在 TestServiceImpl 业务层中，将通过 TestManager 管理器得到一个 ChaincodeManager 对象，ChaincodeManager 对象可以直接操作智能合约，执行如下两个方法：

1. invoke(String fcn, String[] args)

该方法传入两个参数，参数 1 是智能合约中的方法名，参数 2 是智能合约所需的数据数组。

2. query(String fcn, String[] args)

该方法中的参数含义与上述一样。

根据上述说明，编写 TestServiceImpl 源码：

```
public class TestServiceImpl implements TestService {

    @Override
    public String chaincode(JSONObject json) {
        String fcn = json.getString("fcn");
        JSONObject argJson = json.getJSONObject("arg");
        Map<String, String> resultMap;
        List<String> args = new LinkedList<String>();
        String execCode = "";
        String execResult = "";
        try {
            ChaincodeManager manager =
TestManager.obtain().getDemoChaincodeManager();
            switch (fcn) {
            case "invoke":
                args.add(argJson.has("A") ? argJson.getString("A") : "");
                args.add(argJson.has("B") ? argJson.getString("B") : "");
                args.add(argJson.has("val") ? argJson.getString("val") : "");
                String[] arguments = new String[args.size()];
                args.toArray(arguments);
                resultMap = manager.invoke(fcn, arguments);
                break;
            case "query":
                args.add(argJson.has("A") ? argJson.getString("A") : "");
                arguments = new String[args.size()];
                args.toArray(arguments);
                resultMap = manager.query(fcn, arguments);
                break;
            default:
                return responseFail("No func found, please check and try again.");
            }
            execCode = resultMap.get("code");
            execResult = resultMap.get("data");
            if (execCode.equals("error")) {
```



```
                return responseFail(execResult);
            } else {
                return responseSuccess(execResult);
            }
        } catch (CryptoException e1) {
            e1.printStackTrace();
            return responseFail("请求失败：crypto-config 文件证书异常");
        } catch (InvalidArgumentException e1) {
            e1.printStackTrace();
            return responseFail("请求失败：无效的参数异常");
        } catch (NoSuchAlgorithmException e1) {
            e1.printStackTrace();
            return responseFail("请求失败：算法异常");
        } catch (NoSuchProviderException e1) {
            e1.printStackTrace();
            return responseFail("请求失败：没有授信方异常");
        } catch (InvalidKeySpecException e1) {
            e1.printStackTrace();
            return responseFail("请求失败：规范无效异常");
        } catch (TransactionException e1) {
            e1.printStackTrace();
            return responseFail("请求失败：请求事务异常");
        } catch (IOException e1) {
            e1.printStackTrace();
            return responseFail("请求失败：IO异常");
        } catch (ProposalException e) {
            e.printStackTrace();
            return responseFail("请求失败：提案异常");
        } catch (InterruptedException e) {
            e.printStackTrace();
            return responseFail("请求失败：被打断");
        } catch (ExecutionException e) {
            e.printStackTrace();
            return responseFail("请求失败：执行异常");
        } catch (TimeoutException e) {
            e.printStackTrace();
            return responseFail("请求失败：超时异常");
        } catch (JSONException e) {
            e.printStackTrace();
            return responseFail("请求失败：JSONException");
        }
    }

}
```

在 TestServiceImpl 对象中，接收了智能合约中的 invoke 与 query 方法，启动服务器后可通过 postman 等相关工具直接测试，query 测试如图 9-3 所示。

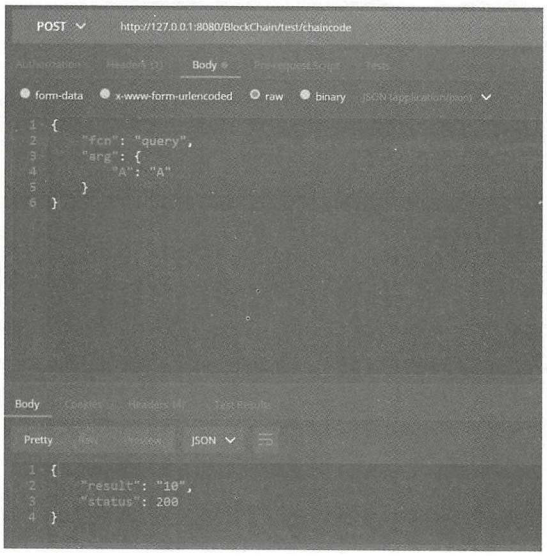


图9-3　query测试

看到该方法可以查出当前 A 的值为 10。继续看 invoke 的结果，如图 9-4 所示。

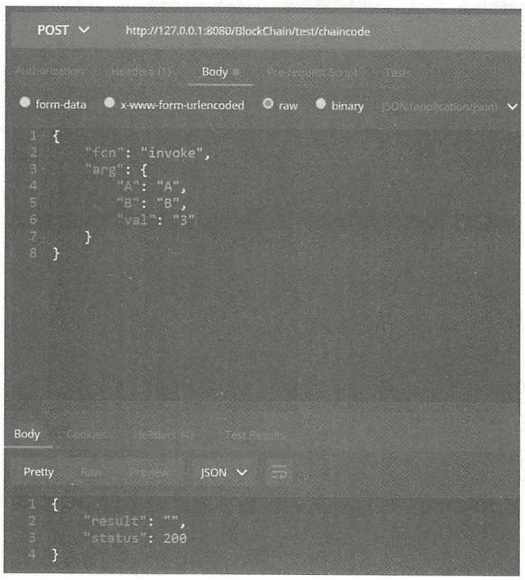


图9-4　invoke测试



该方法将 A 的资产 3 转移到 B 的资产中，结果返回成功，但执行结果需要再次通过 query 执行查询，如图 9-5 所示。

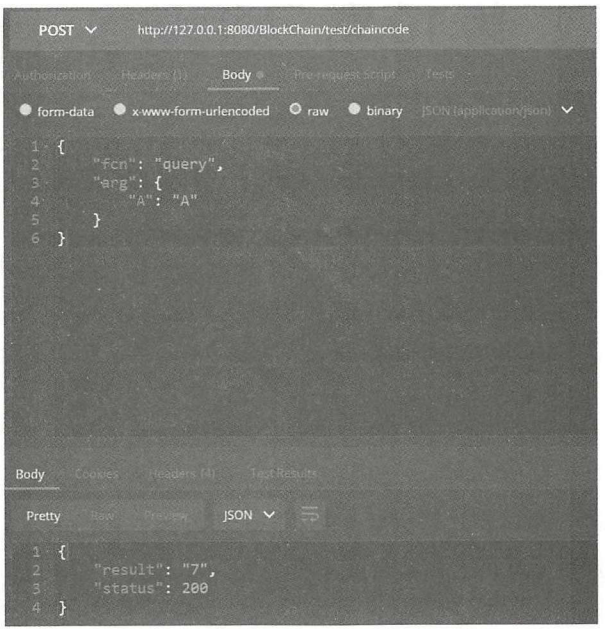


图9-5　执行后结果

这次看到了 A 的资产由 10 变成了 7，说明之前的 invoke 操作成功。

至此，有关 Java-SDK 的内容告一段落。关于 Java-SDK 的内容远不止这些，本书只起到入门的作用，更多有关如何监听区块广播、事件以及内容追溯等方案，在 SDK 中都能找到答案，如果结合 Chaincode API 一起查阅效果会更好。

## 9.4　本章小结

本章主要概述了 Java-SDK 的基本用法，相对于 Fabric-SDK-Java 的官方案例，本章仅是对代码做了简单的处理，使得易读性更强。

对于区块链应用层开发人员而言，如果是 Java 语言使用者，在没有特殊需求的情况下，使用本章解决方案即可实现与区块链底层的交互。另外，深层次研读各个封装出来对象的源码和其中意图，也能够加深对 HyperLedger Fabric 请求事务的理解。

# 第 10 章　项目演练

区块链在本书中指的是类似 HyperLedger Fabric 这样的联盟链或私链平台应用，以炒币概念发展的区块链项目不在本书的讨论范围内。

区块链在实际落地应用中有一个重要的作用，那就是解决交易的信任和安全问题。如何围绕着这一点来切入已有的生产系统中成为现阶段绝大多数专家们探讨的问题。而针对这一问题，区块链自身也具备了四种成熟的技术特性：

1. 分布式账本

即由分布在不同地方的多个节点共同完成交易记账的功能，并且每一个节点都记录了一个完整的备份，它们都可以参与并监督交易合法性。如果参与记账的节点足够多，那么除非所有的节点在同一时间都遭到破坏，否则账目就不会丢失，从而保证了账目数据的安全性。

2. 非对称加密和授权技术

存储在区块链上的交易信息对已加盟成员来说是公开透明的，但是每一个参与成员的信息是高度加密的，只有在联盟授权的情况下才能访问，从而保证了数据的安全和个人的隐私。

3. 共识机制

即所有记账节点之间怎么达成共识，去认定一个记录的有效性，这既是认定的手段，也是防止篡改的手段。基于 HyperLedger Fabric 的联盟区块链，则具有更强制性的要求，即拥有合法背书的成员才能参与账本的数据变更。

4. 智能合约

智能合约是基于这些可信的不可篡改的数据，可以自动化地执行一些预先定义好的规则和条款。它应该与业务尽可能地分离，以便于各个加盟成员通过共享数据实现自身业务。

本书观点是任何项目都不要刻意围绕区块链开展，不能因为区块链当前火爆就强迫业务进行转型，否则对企业发展和项目升级并没有起到积极作用。

区块链应该被打造成一个独立于企业内部核心业务的外围应用。换一种说法，一家企业加盟某一区块链平台的目的是提升自身的业务健壮度和抗风险能力，而不能因为加盟了某一区块

链平台就有针对性地修改了自身原有业务。

企业对区块链联盟平台的姿态应该是最大程度自由的，即上链有一定好处，但依然可以根据自身的实际情况选择随时下链而不影响原有业务。

本章节项目演练以区块链联盟平台发起方的视角来进行，作为发起方也应该考虑到上述所说内容，即不要在区块链联盟平台设计之初就把业务捆绑在上面。客观来讲，最好可以做到业务与数据的分离，在绝大多数前后台业务开发中，这也应该是一种较为主流的思想和做法。

区块链的作用是共享联盟中所有成员都认可的、有价值的数据，至于各家成员如何来使用这些数据完善自身业务，不应该是区块链平台考虑的问题。基于这一开发思想，让区块链平台变得更简单，让智能合约也运转得更高效。

本书的项目演练均以第 6 章的平台为基础进行操作。

## 10.1　反欺诈系统

### 10.1.1　需求分析

反欺诈系统的组成包括数据采集、数据存储、欺诈侦测、系统管理、监控报表等子系统。各子系统协同运作，构建起一个完整的反欺诈系统。通过构建有效的网络反欺诈的方案，可以有效监控互联网中的各种欺诈行为，并减少互联网中企业和用户的财产损失。

传统金融机构，特别是传统小贷公司，在放贷前一定会对客户做大量的背景调查工作。

金融结构可以借助央行的征信报告，或者公司内部的客户信息进行考核。但很多时候，这些报告信息并不全面。

而根据区块链的特性，不难发现这是一个绝佳的解决方案，且完全可以当作企业内部已有的反欺诈系统的外围使用，对已经存在的业务也不会造成侵入式影响。

通过区块链技术，可以使得各金融机构共享所有的用户资产报告，且可以做到任意一家企业只能获取本企业已拥有用户的报告内容，确保非相关用户的资产信息内容不会被恶意泄露。

利用区块链技术同时可以让各金融机构之间的沟通变得更加简单，不再需要相互开发往来接口，也不必为各自用户信息的保密性付出太多的工作，这一切都由区块链的特性合理解决。

这里以小贷金融机构为例，A 用户以资金周转的名义在 M 小贷公司贷款 50 万元，根据 M 公司的审查，认可 A 具备贷款期限内有 50 万元的还款能力。

与此同时，A 用户又以同样的理由在 N 小贷公司贷款 50 万元，这时候 N 公司也做了相同的审查工作。但 N 公司却不知道 A 用户已经在 M 公司贷款了 50 万元，且 A 用户经审查在贷款期限内的还款能力只有 70 万元。

此时，如果 N 公司将钱贷给了 A 用户，极有可能 M 和 N 公司其中一家或两家同时出现逾

贷的状况。

这就是在信息不对称的情况下，A 用户的行为对 M 和 N 公司乃至通过 M 和 N 公司借钱给 A 用户的其他用户造成经济损失。

这仅仅是其中一笔，如果出现大量的上述情况，带来的损失将会直线上升，甚至让一家小贷公司破产，无数个家庭被摧毁。

后果是可怕的，当下迫切需要一套信息透明的分享机制来遏制这样的情况发生。

如果 A 用户已经在 M 公司贷款了 50 万元，那么当 A 用户再去 N 公司贷款时，N 公司通过区块链联盟平台能够查询到 A 用户在 M 公司的贷款记录，且经过审查判断出 A 用户的还款能力只有 70 万元。那么，N 公司可以针对 A 用户的贷款进行 30 万元以下的审批，确保出现的逾贷损失程度为最低限度，保护公司及投资用户的切身利益。

### 10.1.2 编写合约

针对上述需求，这里需要编写一套符合数据需求的智能合约。

因为是金融类项目，所以合约入口将是一个以 finance 命名的 Go 文件。

接下来，计划编写一个名为 user 的 Go 文件，里面承载了用户的相关信息和用户方法，先从用户的行为来具体分析业务的可能性。

用户应该具备一些最基本的属性，比如用户名与身份证号等信息，但这些信息是不应该被读取的，所以用户的唯一 ID 应该是姓名与身份证号组合后的 MD5。这样的设计可以防止已经加盟的小贷公司通过手上的身份证号来恶意刷区块链平台中的用户信息，通过姓名与身份证号的组合，可以有效防止被刷。毕竟从公司体量及业务方向来看，获取无数个身份证号或许很容易，但要使这些身份证号对应上姓名，难度就很大了。

当可以确认用户唯一性后，可以凭借用户 ID 对该用户的行为进行写入和读取操作。但如果真的以用户为中心进行后续操作，那将会遇到另外一种极为小概率问题，即有可能多家公司同时对同一用户进行信息写入操作，就会触发事务问题。

参考第 6.3 节讲到的读写集部分，其中概述了读写集的具体内部状况。如果同时存在多个写集，且写集的读集版本号一致，则会导致后进来的写集操作失败。目前 HyperLedger Fabric 并不支持多个写集同时操作。

除了上述小概率原因外，还牵涉到一个背书和数据虚假处理的问题。

如果区块链联盟中有一个成员因竞争或自身系统 Bug 等问题，导致向区块链联盟平台输入了大量错误的且几乎无法追溯源头的用户数据，从而直接导致整个区块链联盟平台的数据出现严重错误。即如果 A 用户本身还有 80 万元的贷款额度，却因为数据的问题，导致 A 用户显示的贷款额度为负值，这不仅损害了用户的利益，更是直接对加盟联盟的所有企业带来巨大损失。

从上述观点出发，就不能编写以用户为核心的智能合约，而用户 ID 的唯一性又必不可缺。那么换一个角度，可以以用户与小贷公司的合同作为合约核心。

现在开始编写一个名为 Compact 的 Go 文件，它是合同对象，拥有一些合同所必备的基本属性，以及签署该合同的用户 ID。所以，在 Compact 中会有一个名为 Compact 的 struct 来存储这些属性，如下所示：

```
// 合同全集详情
// 本条记录主键key由成员ID和合同ID联合组成，具备唯一性
type Compact struct {
 Timestamp        int64       `json:"timestamp"`          // 本条记录创建时间戳
 UID              string      `json:"uid"`                // 用户唯一ID
 LoanAmount       string      `json:"loanAmount"`         // 用户贷款金额
 ApplyDate        string      `json:"applyDate"`          // 申请日期
 CompactStartDate  string      `json:"compactStartDate"`    // 贷款开始日期
 CompactEndDate    string      `json:"compactEndDate"`     // 贷款计划终止日期
 RealEndDate      string      `json:"realEndDate"`        // 贷款实际终止日期
}
```

该 Compact 会有其对应生成 Json 对象的 key 属性，以便于在使用 CouchDB 的时候更好地支持富查询。

此时已经有了“合同”对象，接下来需要实现记录合同数据以及根据用户 ID 取出与之对应的合同数组。先编写记录合同数据的方法，如下所示：

```
// 贷款操作
// args: UID、贷款金额、申请日期、贷款开始日期、贷款计划终止日期、合同ID
// name: 成员名称
func Loan(stub shim.ChaincodeStubInterface, args []string, name string) error {
 if len(args) != 6 {
     return fmt.Errorf("Parameter count error while Loan, count must 5")
 }
 if len(args[0]) != 32 {
     return fmt.Errorf("Parameter uid length error while Loan, 32 is right")
 }
 if len(args[2]) != 14 {
     return fmt.Errorf("Parameter ApplyDate length error while Loan, 14 is right")
 }
 if len(args[3]) != 14 {
     return fmt.Errorf("Parameter CompactStartDate length error while Loan, 14 is
right")
 }
 if len(args[4]) != 24 {
     return fmt.Errorf("Parameter CompactEndDate length error while Loan, 14 is
right")
```

```
    }
    var compact Compact
    compact.Uid = args[0]
    compact.LoanAmount = args[1]
    compact.ApplyDate = args[2]
    compact.CompactStartDate = args[3]
    compact.CompactEndDate = args[4]
    compact.Timestamp = time.Now().Unix()

    compactJsonBytes, err := json.Marshal(&compact) // Json序列化
    if err != nil {
        return "", fmt.Errorf("Json serialize Compact fail while Loan, compact id = "
+ args[5])
    }
    // 生成合同联合主键
    key, err := stub.CreateCompositeKey("Compact", []string{name, args[5]})
    if err != nil {
        return "", fmt.Errorf("Failed to CreateCompositeKey while Loan")
    }
    // 保存合同信息
    err = stub.PutState(key, compactJsonBytes)
    if err != nil {
        return "", fmt.Errorf("Failed to PutState while Loan, compact id = " +
args[5])
    }
   }
```

在 Loan 方法中接收了 6 个参数，方法初始做了一些可自定义的简单判断，并以此传入 UID、贷款金额、申请日期、贷款开始日期、贷款计划终止日期及合同 ID。其中，成员名称及合同 ID 并不是 Compact 对象的属性，也不会存入区块的 value 中，它用来做 Compact 的组合键，确保存入 key 的唯一性，也便于存入该数据的成员自行校验已存数据。

完整的 compact.go 文件源码如下：

```
package bean

import (
 "encoding/json"
 "fmt"
 "time"

 "github.com/hyperledger/fabric/core/chaincode/shim"
)

// 合同全集详情
```

```
    // 本条记录主键key由成员ID和合同ID联合组成，具备唯一性
    type Compact struct {
     Timestamp       int64  `json:"timestamp"`        // 本条记录创建时间戳
     Uid             string `json:"uid"`             // 用户唯一ID（32位MD5值）
     LoanAmount      string `json:"loanAmount"`       // 用户贷款金额
     ApplyDate       string `json:"applyDate"`        // 申请日期
     CompactStartDate string `json:"compactStartDate"` // 贷款开始日期
     CompactEndDate   string `json:"compactEndDate"`   // 贷款计划终止日期
     RealEndDate     string `json:"realEndDate"`      // 贷款实际终止日期
    }

    // 贷款操作
    // args: UID、贷款金额、申请日期、贷款开始日期、贷款计划终止日期、合同ID
    // name: 成员名称
    func Loan(stub shim.ChaincodeStubInterface, args []string, name string) error {
     if len(args) != 6 {
         return fmt.Errorf("Parameter count error while Loan, count must 5")
     }
     if len(args[0]) != 32 {
         return fmt.Errorf("Parameter uid length error while Loan, 32 is right")
     }
     if len(args[2]) != 14 {
         return fmt.Errorf("Parameter ApplyDate length error while Loan, 14 is right")
     }
     if len(args[3]) != 14 {
         return fmt.Errorf("Parameter CompactStartDate length error while Loan, 14 is
right")
     }
     if len(args[4]) != 24 {
         return fmt.Errorf("Parameter CompactEndDate length error while Loan, 14 is
right")
     }
     var compact Compact
     compact.Uid = args[0]
     compact.LoanAmount = args[1]
     compact.ApplyDate = args[2]
     compact.CompactStartDate = args[3]
     compact.CompactEndDate = args[4]
     compact.Timestamp = time.Now().Unix()

     compactJsonBytes, err := json.Marshal(&compact) // Json序列化
     if err != nil {
         return fmt.Errorf("Json serialize Compact fail while Loan, compact id = " +
args[5])
```

```
    }
    // 生成合同联合主键
    key, err := stub.CreateCompositeKey("Compact", []string{name, args[5]})
    if err != nil {
        return fmt.Errorf("Failed to CreateCompositeKey while Loan")
    }
    // 保存合同信息
    err = stub.PutState(key, compactJsonBytes)
    if err != nil {
        return fmt.Errorf("Failed to PutState while Loan, compact id = " + args[5])
    }
    return nil
}
```

compact.go 文件位于与 finance.go 文件同级 bean 目录下。

在 compact 文件中的 Loan 方法需要传入一个成员名称的字符串，该字符串并非接口传入，即非应用层协助，而是通过 Chaincode API 自动获取，所以需要编写一个 utils.go 的工具类，主要用来服务主函数调用。

utils.go 工具类源码如下：

```
package utils

import (
    "fmt"
    "bytes"
    "crypto/x509"
    "encoding/pem"
    "strings"

    "github.com/hyperledger/fabric/core/chaincode/shim"
)

// 获取当前操作智能合约成员的具体名称，如a1aw28
func GetCreatorName(stub shim.ChaincodeStubInterface) (string, error) {
    name, err := GetCreator(stub) // 获取当前智能合约操作成员名称
    if err != nil {
        return "", err
    }
    // 格式化当前智能合约操作成员名称
    memberName := name[(strings.Index(name, "@") + 1):strings.LastIndex(name,
".example.com")]
    return memberName, nil
}
```

```
// 获取操作成员
func GetCreator(stub shim.ChaincodeStubInterface) (string, error) {
 creatorByte, _ := stub.GetCreator()
 certStart := bytes.IndexAny(creatorByte, "-----BEGIN")
 if certStart == -1 {
     fmt.Errorf("No certificate found")
 }
 certText := creatorByte[certStart:]
 bl, _ := pem.Decode(certText)
 if bl == nil {
     fmt.Errorf("Could not decode the PEM structure")
 }

 cert, err := x509.ParseCertificate(bl.Bytes)
 if err != nil {
     fmt.Errorf("ParseCertificate failed")
 }
 uname := cert.Subject.CommonName
 return uname, nil
}
```

接下来编写 finance.go 文件，finance 是整个智能合约的入口，它的 Init 方法中并不需要初始化任何数据。它需要一个 loan 方法执行贷款记录操作，可以参考第 8 章中的 marbles02 示例写一个 query 方法来实现富查询。这里将换一种操作思路，采用对 CouchDB 直连的方式读取数据。

finance.go 源码如下：

```
// finance
package main

import (
 "fmt"

 "github.com/hyperledger/fabric/core/chaincode/shim"
 "github.com/hyperledger/fabric/protos/peer"

 "github.com/hyperledger/fabric/aberic/chaincode/go/finance/bean"
 "github.com/hyperledger/fabric/aberic/chaincode/go/finance/utils"
)

type Finance struct {
}

func (t * Finance) Init(stub shim.ChaincodeStubInterface) peer.Response {
```

```
    args := stub.GetStringArgs()
    if len(args) != 0 {
        return shim.Error("Parameter error while Init")
    }
    return shim.Success(nil)
}

func (t * Finance) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    fn, args := stub.GetFunctionAndParameters()
    switch fn {
    case "loan": // 记录贷款数据
        return loan(stub, args)
    default:
        return shim.Error("Unknown func type while Invoke, please check")
    }
}

// 记录贷款数据
func loan(stub shim.ChaincodeStubInterface, args []string) peer.Response {
    name, err := utils.GetCreatorName(stub)
    if err != nil {
        return shim.Error(err.Error())
    }

    err = bean.Loan(stub, args, name)
    if err != nil {
        return shim.Error(err.Error())
    }

    return shim.Success([]byte("记录贷款数据成功"))
}

func main() {
    if err := shim.Start(new(Finance)); err != nil {
        fmt.Printf("Chaincode startup error: %s", err)
    }
}
```

接下来便是线上验证该合约是否运行成功且符合需求。

### 10.1.3 线上验证

该合约是一个名为 finance 目录的包，结构如图 10-1 所示。



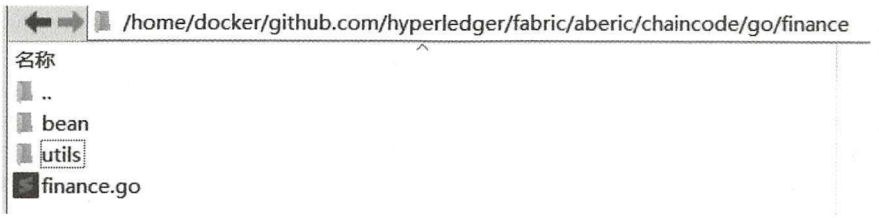


图10-1　finance智能合约

将该合约上传至 172.31.159.129 服务器的/home/docker/github.com/hyperledger/fabric/Aberic/chaincode/go 目录下，bean 目录下是 compact.go 对象，utils 目录下是 utils.go 对象。

因为该项目依赖于第 6 章的配置，即可在 mychannel 上也安装该合约。而该合约在编写时的 Init 方法并不需要初始化参数，根据实际需求在 mychannel 上安装并实例化 finance 合约，这些操作参考第 6 章即可。

下面开始执行记录贷款信息的操作，具体执行如下命令：

```
    peer chaincode invoke -C mychannel -n mychannel -v 1.0 -c '{"Args":["loan",
    "4e4d6c332b6fe62a63afe56171fd3725", "200000", "20180420080812", "20180421080812",
"20190421080812", "000001"]}'
```

根据合约，该命令的意思是一个被加密后为 4e4d6c332b6fe62a63afe56171fd3725 的用户于 2018 年 04 月 20 日 08 时 08 分 12 秒申请了一笔 20 万元的贷款，贷款起止日期分别是 2018 年 04 月 21 日 08 时 08 分 12 秒和 2019 年 04 月 21 日 08 时 08 分 12 秒，该笔贷款合同 ID 为 000001。

执行后有如下日志则表示执行成功：

```
    2018-04-21 08:58:16.365 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 06f
Using
     default escc
    2018-04-21 08:58:16.365 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 070
Using default vscc
    2018-04-21 08:58:16.365 UTC [chaincodeCmd] getChaincodeSpec -> DEBU 071 java
chaincode disabled
    2018-04-21 08:58:16.365 UTC [msp/identity] Sign -> DEBU 072 Sign: plaintext:
0AAC070A6C08031A0C08A8FAEBD60510...32313038303831320A06303030303031
    2018-04-21 08:58:16.365 UTC [msp/identity] Sign -> DEBU 073 Sign: digest:
603E7CF05C25AE09F071CE28B8F85A2D2EDEA33DC101B5CB7710ECF7477E6C9B
    2018-04-21 08:58:16.390 UTC [msp/identity] Sign -> DEBU 074 Sign: plaintext:
0AAC070A6C08031A0C08A8FAEBD60510...3A643928FDC9A181F78C8E8B5C5D76F3
    2018-04-21 08:58:16.390 UTC [msp/identity] Sign -> DEBU 075 Sign: digest:
7ABD5501BCA96B5B594A8E79CEFFB52B36AF4DD368B8289DBCE0FC488D4E02D6
    2018-04-21 08:58:16.393 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 076 ESCC
```

```
invoke result: version:1 response:<status:200 message:"OK" payload:"\350\256\260\345\
275\225\350\264\267\346\254\276\346\225\260\346\215\256\346\210\220\345\212\237" >
payload:"\n J\333\033%\245\225\241\234\331\315\265E\310\263K\356\000\323\356\210Af`\
3031d\272\377\374\232[[\022\310\002\n\224\002\022\031\n\004lscc\022\021\n\017\n\tmycha
nnel\022\002\010\002\022\366\001\n\tmychannel\022\350\001\032\345\001\n\025\000Compact
\000org1\000000001\000\032\313\001{\"timestamp\":1524301096,\"uid\":\"4e4d6c332b6fe62a
63afe56171fd3725\",\"loanAmount\":\"200000\",\"applyDate\":\"20180420080812\",\"compac
tStartDate\":\"20180421080812\",\"compactEndDate\":\"20190421080812\",\"realEndDate\":
\"\"}\032\035\010\310\001\032\030\350\256\260\345\275\225\350\264\267\346\254\276\346\
225\260\346\215\256\346\210\220\345\212\237\"\020\022\tmychannel\032\0031.5"
endorsement:<endorser:"\n\007Org1MSP\022\222\006-----BEGIN CERTIFICATE-----
\nMIICGDCCAb+gAwIBAgIQEsKBEKwwwpp/z/Xn5gLApDAKBggqhkjOPQQDAjBzMQsw\nCQYDVQQGEwJVUzETMB
EGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMNU2FuIEZy\nYW5jaXNjbzEZMBcGA1UEChMQb3JnMS5leGFtcG
xlLmNvbTEcMBoGA1UEAxMTY2Eu\nb3JnMS5leGFtcGxlLmNvbTAeFw0xODA0MTEwMzU0MTZaFw0yODA0MDgwMz
U0MTZa\nMFsxCzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpDYWxpZm9ybmlhMRYwFAYDVQQHEw1T\nYW4gRnJhbmNp
c2NvMR8wHQYDVQQDExZwZWVyMC5vcmcxLmV4YW1wbGUuY29tMFkw\nEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE
Cpresx/Xv/LbmL7P0hPrngE603IoYKOj\n2E22NnW7tVdr0XtQJSXMOaBLHoPU7P0mBv8xcqBirmjBp463axjY
AKNNMEswDgYD\nVR0PAQH/BAQDAgeAMAwGA1UdEwEB/wQCMAAwKwYDVR0jBCQwIoAgq33KXl9rHMJM\nICN2TF
s00feNhhTSoR50F40tVQm9O+gwCgYIKoZIzj0EAwIDRwAwRAIgLpvW805T\n5IpTQlYSLTs+T7eFPm37rzVu2l
lABopwOWECIE4clYkh7ILEmxytYE2Vvtun4P4D\naGUb2JbPjKHxUnnL\n-----END CERTIFICATE-----\n"
signature:"0D\002 Lg\260\342\025\366\260\201\206\250\272x\363\247\010AUJ\323\033\006
\300t\246\0208\300^\031\321\237\342\002 \022\355:\327\016\0349\343\267\251\335\253Lp\
344\260:d9(\375\311\241\201\367\214\216\213\\]v\363" >
    2018-04-21 08:58:16.394 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 077
Chaincode invoke successful. result: status:200 payload:"\350\256\260\345\275\225
\350\264\267\346\254\276\346\225\260\346\215\256\346\210\220\345\212\237"
    2018-04-21 08:58:16.394 UTC [main] main -> INFO 078 Exiting.....
```

接下来，可以对同一用户多次执行借贷操作，执行如下命令：

```
    peer chaincode invoke -C mychannel -n mychannel -v 1.0 -c '{"Args":["loan",
    "4e4d6c332b6fe62a63afe56171fd3725", "200000", "20180420080812", "20180421080812",
"20190421080812", "000002"]}'

    peer chaincode invoke -C mychannel -n mychannel -v 1.0 -c '{"Args":["loan",
    "4e4d6c332b6fe62a63afe56171fd3725", "200000", "20180420080812", "20180421080812",
"20190421080812", "000003"]}'
```

即该用户总计发起了三次贷款，每笔贷款金额为 20 万元，最终贷款金额为 60 万元。最终可通过 CouchDB 后台查看数据记录，如图 10-2 所示。



| _id | applyDate | compactEnd... | compactStart... | loanAmount |
| --- | --- | --- | --- | --- |
| □Compact□org1□000001□ | 20180420080812 | 20190421080812 | 20180421080812 | 200000 |
| □Compact□org1□000002□ | 20180420080812 | 20190421080812 | 20180421080812 | 200000 |
| □Compact□org1□000003□ | 20180420080812 | 20190421080812 | 20180421080812 | 200000 |

图10-2　三笔贷款记录

接下来开始使用这些数据。因为要根据用户唯一 ID 查询该用户下的所有贷款记录，而用户唯一 ID 在之前的分析中提到并不适合做 key，只能用于贷款合同的值存在。根据值查询数据需要用到富查询，参考第 8 章中提到的方案，建立如下索引：

```
{
 "index": {
     "fields": [
              "uid"
]
 },
 "name": "indexUid",
 "ddoc": "indexUidDoc",
 "type": "json"
}
```

随后可在 CouchDB 后台看到如图 10-3 所示索引记录。

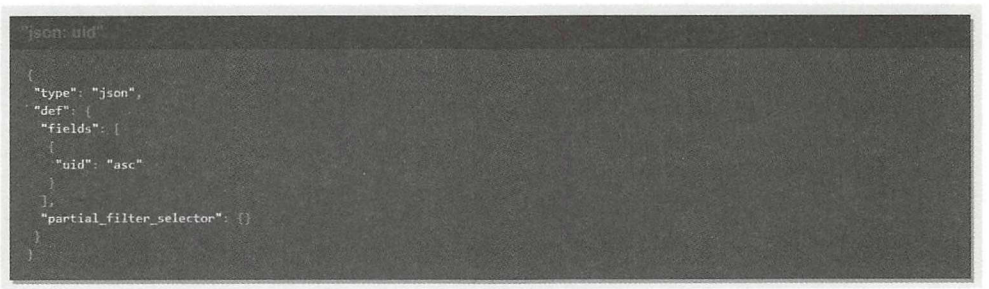


图10-3　索引记录

索引建立成功后，可以先在 CouchDB 后台中执行一个 Mango Query 进行测试，测试 json 如下：

```
{
  "selector": {
     "uid": "4e4d6c332b6fe62a63afe56171fd3725"
  }
```

```
}
```

执行上述 json 后，可以在右侧得到查询结果，如图 10-4 所示。

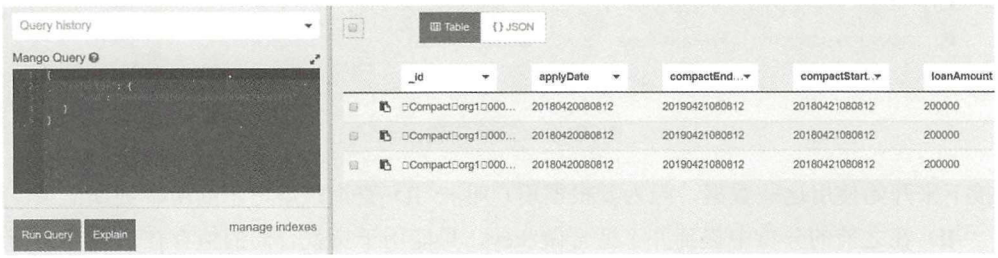


图10-4　索引查询

在生产实际应用中，理论上执行读写集查询时必须经过智能合约。但如果仅仅是用作读集，可以通过 CouchDB 直连的方式执行，所以在本章中的智能合约并没有编写 query 方法。

**注意**：这不是正规的架构思路，仅仅是演练实际应用中的场景方案。

应用层如 Java、py、PHP 等服务可以通过 http://IP:PORT/dbname/_find 结果，直接 Post 上述查询 json 即可得到结果。如本节例子中将上述 selector 请求 json 通过 Post 发送给 http://172.31.159.129:5984/mychannel_finance/_find，有如下所示结果：

```
{
    "docs": [
        {
            "_id": "\u0000Compact\u0000org1\u0000000000001\u0000",
            "_rev": "1-3a2bc44e1f867f435a075cc9b411199d",
            "applyDate": "20180420080812",
            "compactEndDate": "20190421080812",
            "compactStartDate": "20180421080812",
            "loanAmount": "200000",
            "realEndDate": "",
            "timestamp": 1524301096,
            "uid": "4e4d6c332b6fe62a63afe56171fd3725",
            "~version": "3:0"
        },
        {
            "_id": "\u0000Compact\u0000org1\u0000000000002\u0000",
            "_rev": "1-ffc6a58d0ae883c7f23a913d92743e17",
            "applyDate": "20180420080812",
            "compactEndDate": "20190421080812",
            "compactStartDate": "20180421080812",
```



```
            "loanAmount": "200000",
            "realEndDate": "",
            "timestamp": 1524301531,
            "uid": "4e4d6c332b6fe62a63afe56171fd3725",
            "~version": "4:0"
        },
        {
            "_id": "\u0000Compact\u0000org1\u0000000000003\u0000",
            "_rev": "1-e24b16eab9c7351b129983178dd6d39b",
            "applyDate": "20180420080812",
            "compactEndDate": "20190421080812",
            "compactStartDate": "20180421080812",
            "loanAmount": "200000",
            "realEndDate": "",
            "timestamp": 1524301535,
            "uid": "4e4d6c332b6fe62a63afe56171fd3725",
            "~version": "5:0"
        }
    ],
    "bookmark":
"g1AAAABreJw1yjsSgCAMANFIZWVn7xGEaDyAB3GQgD8cHKTx9krhljvPA4BYBUN1Jx3TYZ-
Jg9n4_H4NYzgvbRKEuEhocwgZlz_2HyuybTrbMRlENZOzpDShdrYnOUjHOKh-31_dSxvm"
}
```


## 10.3　本章小结

本章的应用尚不能实际用于生产，主要目的是提供一种落地实践的经验，通过对实际业务和需求进行针对性编写智能合约，并结合最终业务对值的索取需求采用 CouchDB 作为底层数据库，并编写与业务对应的索引实现取值操作。

在最后实现取值操作的过程中，本章在合约中并未编写实际获取方法。如果在单机链的情况下，可以通过这样的方式实现获取区块链中数据的目的。但在联盟链中，必须为所有的加盟方提供开源可见的获取方法，并在智能合约中体现。

## 内容简介

本书系统地介绍了超级账本HyperLedger Fabric v1.1架构的设计和应用方法，包括环境及源码部署、Solo多机部署、Kafka集群部署、智能合约编写等。同时，针对第三方可插拔式插件CouchDB实战应用，Java-SDK的应用、编写方案和具体接口执行策略进行了详细讲解。另外，本书以搭建一个反欺诈区块链平台项目为例进行了实战演练，读者可以快速掌握区块链技术。

本书适合区块链系统开发人员阅读，需要有一定的面向对象语言的基础，也可供对开发区块链系统感兴趣的高校师生参考。

欢迎投稿：
邮箱：syd@phei.com.cn
微信：qie_yd

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版


上架建议：区块链

ISBN 978-7-121-34173-1

9 787121 341731 >

定价：79.00元

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

责任编辑：宋亚东
封面设计：李　玲